

# PROGRAM SYNTHESIS BENCHMARK FOR VISUAL PROGRAMMING IN XLOGOONLINE ENVIRONMENT

**Anonymous authors**

Paper under double-blind review

## A TABLE OF CONTENTS

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

- Appendix B provides more details about the datasets.
- Appendix C provides more details about the fine-tuning process and evaluation.
- Appendix D provides additional experiments and results.
- Appendix E provides more details about the prompts used for fine-tuning and evaluation.

## B MORE DETAILS ABOUT THE DATASETS

We provide the following details about the datasets.

1. *Real-world tasks in the XLogoOnline platform:* The real-world visual programming tasks in the REAL 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 8 shows the screenshots of the platform.
2. *The benchmark source code and datasets:* See the provided file **iclr2025-xlogo-benchmark\_src.zip**. After unzipping this file, the dataset is available in the *data.zip* file.
3. *Data license confirmation:* We confirm that all data used in this paper is either publicly available or has been obtained and used in accordance with the relevant data licenses.

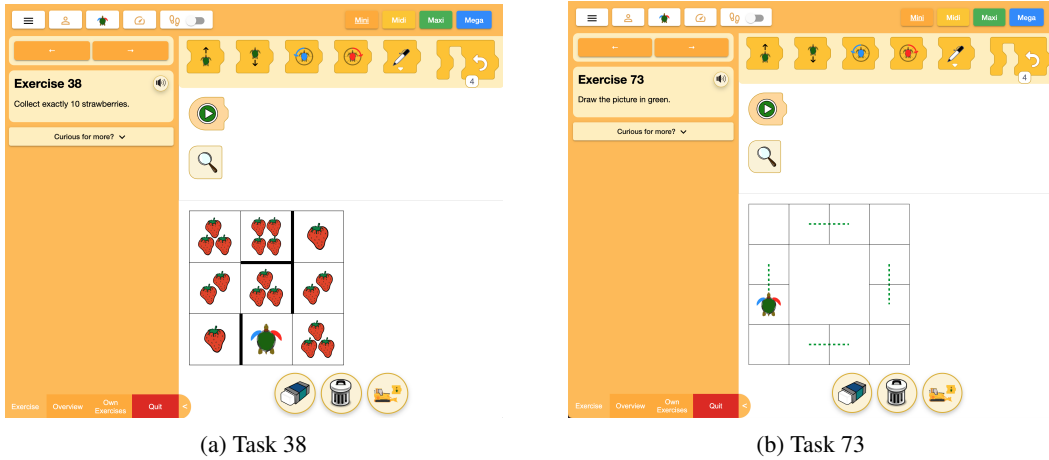


Figure 8: Example tasks from the XLogoOnline platform. Students need to drag and drop different blocks to solve the tasks.

## B.1 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 REAL dataset, resulting in a set  $\{(T_i, C_i)\}_{i=1}^N$ . 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 difficulty level  $D$ . 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  $T$ .
- **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  $(T, C, D)$ , 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, respectively, each containing placeholders. These placeholders are then populated with specific values using an SMT-based constraint solver (de Moura & 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  $(T, C, D)$ , resulting in up to  $500 \text{ tasks} \times 3 \text{ difficulty levels} = 1,500$  tasks for each reference input  $(T, C)$ . 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.

## B.2 QUALITY OF THE DATASETS

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

1. **REAL dataset (85 samples):** This dataset was derived from the visual programming platform XLogoOnline. The tasks included in this platform 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. **SIM-EVAL 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 REAL dataset from this synthetic 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 REAL dataset and randomly sample 5% of tasks from the SIM-EVAL 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.
- 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.

	Visual Appeal	Grid Elements Utility	Code Quality	Overall Quality
REAL	1.00	1.00	1.00	1.00
SIM-EVAL	0.97	0.94	0.89	0.84

Figure 9: Quality annotation results for REAL and SIM-EVAL datasets. For REAL, we annotate all 85 samples, while for SIM-EVAL, we randomly sample 5% of the dataset for annotation.

The results in Figure 9 demonstrate that the overall quality of the REAL dataset is excellent. The SIM-EVAL 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.

## C MORE DETAILS OF THE FINE-TUNING AND EVALUATION

**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  $\alpha$  four times the rank in each case. We found that a rank 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  $1 \times 10^{-4}$ . 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

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

Figure 10: Success rates (%) of GPT-4 and GPT-4V with different prompting strategies on the REAL dataset. 3-shot prompting is not notably effective, but when combined with CoT, it leads to performance improvements. However, for GPT-4V, the vanilla prompt is the most effective.

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 to Llava1.5-13B (Liu et al., 2023). To do this, we follow the default fine-tuning setup and code provided by the authors.<sup>1</sup> 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  $2 \times 10^{-4}$ , 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.

**Details of emulator-driven fine-tuning.** For emulator-driven fine-tuning, we use the same hyperparameters and setup as the 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  $\beta = 1$ .

**Details of evaluation.** To evaluate GPT family models, we use the OpenAI API with a temperature of 0. For Llama3-8B, Llama2-7B, 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. After inference, we use the emulator to evaluate the models’ success rates over the evaluation datasets.

## D ADDITIONAL EXPERIMENTS AND RESULTS

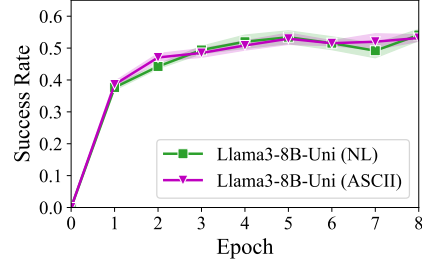
In this section, we present additional experiments and results. First, we investigate the influence of different prompting strategies on model performance. Next, we investigate task representations, comparing natural language and ASCII-based prompts. Then, we analyze the performance of fine-tuned Llama models across different epochs. Finally, we present a case study on output code analysis for perturbed tasks, providing further insights into failure analysis.

### D.1 INFLUENCE OF THE PROMPTING STRATEGIES

Carefully designed prompts have been shown to improve the performance of large models (Wei et al., 2022; Brown et al., 2020). 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, draw, basic

<sup>1</sup><https://github.com/haotian-liu/LLaVA>

	Success Rates (%)	
	NL	ASCII
<b>Base models</b>		
GPT-4	12.94	5.88
Llama3-70B	2.35	1.18
<b>Fine-tuned models</b>		
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 11: 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.

actions, variables, loops, and code constraints. These same 3-shot examples are used to prompt all tasks for *3-shot* and *3-shot + CoT* prompting.

The results are shown in Figure 10. 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.

## D.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 15.

The evaluation results are shown in Figure 11. 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 11b, 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-tuning models in our visual programming domain.

## D.3 FINE-TUNING PERFORMANCE ACROSS DIFFERENT EPOCHS.

Figure 12a 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

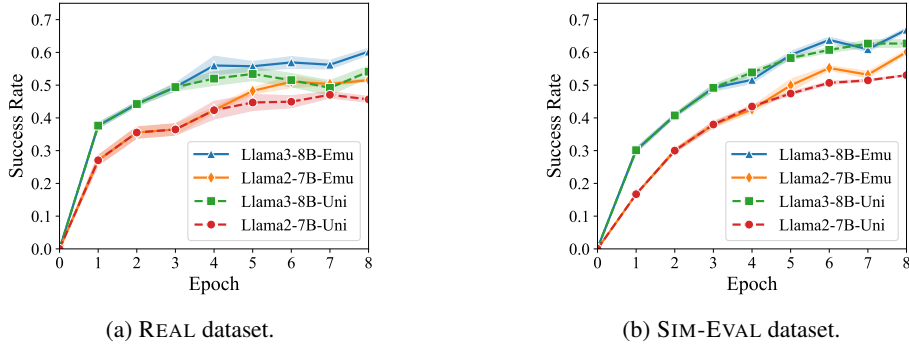


Figure 12: Fine-tuning performance across different epochs on two evaluation datasets. (a) shows the performance of fine-tuned models across different epochs on the evaluation dataset REAL. (b) shows the fine-tuning performance across different epochs on the synthetic evaluation dataset SIM-EVAL.

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 12b, we show the fine-tuning performance across different epochs on the synthetic evaluation dataset SIM-EVAL. 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 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 the resampling models outperforms that of the standard fine-tuning models, indicating the effectiveness of emulator-driven fine-tuning in improving fine-tuning performance.

#### D.4 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 13. 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  $T_{A,B}$ , 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 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.

## E PROMPTS FOR THE XLOGOONLINE-MINI DOMAIN

In this section, we present three types of prompts for program synthesis in the XLogoOnline-Mini domain. Figures 14 and 15 show examples of the prompts using natural language and ASCII representation, respectively. Figure 16 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.

## REFERENCES

- Umair Z. Ahmed, Maria Christakis, Aleksandr Efremov, Nigel Fernandez, Ahana Ghosh, Abhik Roychoudhury, and Adish Singla. Synthesizing Tasks for Block-based Programming. In *NeurIPS*, 2020.
- Tom B. Brown et al. Language Models are Few-Shot Learners. In *NeurIPS*, 2020.
- Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, 2008.
- Juraj Hromkovic, Giovanni Serafini, and Jacqueline Staub. XLogoOnline: A Single-Page, Browser-Based Programming Environment for Schools Aiming at Reducing Cognitive Load on Pupils. In *ISSEP*, 2017.
- Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-Rank Adaptation of Large Language Models. In *ICLR*, 2022.
- Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *SIGOPS*, 2023.
- Haotian Liu, Chunyuan Li, Yuheng Li, and Yong Jae Lee. Improved baselines with visual instruction tuning. *CoRR*, abs/2310.03744, 2023.
- Jacqueline Staub. Logo Environments in the Focus of Time. *Bulletin of EATCS*, 2021.
- Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *NeurIPS*, 2022.
- Chao Wen, Ahana Ghosh, Jacqueline Staub, and Adish Singla. Task Synthesis for Elementary Visual Programming in XLogoOnline Environment. In *AIED Track on Late Breaking Results*, 2024.

Task	GPT-4V	Llama3-70B	Llama3-8B-Uni
Find the strawberry. Use at most 8 commands.  T	<pre>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()</pre>	<pre>def run():     move_forward()     turn_right()     move_forward()     turn_left()     move_forward()     turn_right()     move_forward()     move_forward()</pre>	<pre>def run():     move_forward()     move_forward()     turn_right()     for i in range(3):         move_forward()     turn_right()     for i in range(3):         move_forward()</pre>
Find the strawberry.  T <sub>A</sub>	<pre>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()</pre>	<pre>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()</pre>	<pre>def run():     turn_right()     move_forward()     turn_right()     turn_left()     move_forward()     move_forward()</pre>
Find the strawberry. Use at most 8 commands.  T <sub>B</sub>	<pre>def run():     turn_right() # Turtle faces east     for i in range(3):         move_forward() # Moves from (0,3) to (3,3)</pre>	<pre>def run():     for i in range(3):         move_backward()     turn_right()     move_forward()     turn_left()     move_forward()     turn_right()     move_forward()     move_forward()</pre>	<pre>def run():     move_forward()     turn_right()     for i in range(3):         move_forward()     turn_right()     move_backward()     move_forward()     for i in range(3):         move_forward()</pre>
Find the strawberry. Use at most 8 commands.  T <sub>C</sub>	<pre>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</pre>	<pre>def run():     turn_left()     move_forward()     move_forward()     turn_right()     move_forward()     move_forward()     turn_left()     move_forward()     move_forward()</pre>	<pre>def run():     turn_right()     for i in range(3):         move_forward()     turn_right()     move_forward()     move_forward()</pre>
Find the strawberry.  T <sub>A,B</sub>	<pre>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()</pre>	<pre>def run():     for i in range(3):         move_forward()     turn_right()     move_forward()     turn_left()     move_forward()     for i in range(3):         move_forward()</pre>	<pre>def run():     move_forward()     turn_right()     move_forward()     turn_right()     move_forward()     move_forward()</pre>
Find the strawberry. Use at most 8 commands.  T <sub>B,C</sub>	<pre>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()</pre>	<pre>def run():     turn_left()     turn_left()     for i in range(2):         move_forward()     turn_left()     move_forward()     turn_right()     turn_left()     move_forward()</pre>	<pre>def run():     move_forward()     turn_right()     move_forward()     for i in range(2):         move_forward()     turn_right()     turn_left()     move_forward()</pre>
Find the strawberry.  T <sub>A,C</sub>	<pre>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()</pre>	<pre>def run():     move_forward()     turn_right()     move_forward()     turn_left()     move_forward()     turn_right()     move_forward()     turn_left()     move_forward()     turn_right()     ... (repeat outputs)</pre>	<pre>def run():     move_forward()     turn_right()     move_forward()     turn_right()     move_forward()     move_forward()     turn_left()     move_forward()     turn_right()     move_forward()</pre>
Find the strawberry.  T <sub>A,B,C</sub>	<pre>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()</pre>	<pre>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()</pre>	<pre>def run():     turn_right()     move_forward()     turn_right()     move_forward()     move_forward()     turn_left()     move_forward()     move_forward()</pre>

Figure 13: 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<sub>A</sub>), removing grid constraints (T<sub>B</sub>), simplifying spatial relationships (T<sub>C</sub>), and combinations of these perturbations (T<sub>A,B</sub>, T<sub>B,C</sub>, T<sub>A,C</sub>, and T<sub>A,B,C</sub>). 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.



## 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 14: 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 15: 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 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 16: 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 17: 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 18: 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 19: The detail of the third few-shot + CoT example {few\_shot\_example\_3}.