Neural Task Synthesis for Visual Programming

Anonymous authors
Paper under double-blind review

Abstract

Generative neural models hold great promise in enhancing programming education by synthesizing new content. We seek to design neural models that can automatically generate programming tasks for a given specification in the context of visual programming domains. Despite the recent successes of large generative models like GPT-4, our initial results show that these models are ineffective in synthesizing visual programming tasks and struggle with logical and spatial reasoning. We propose a novel neuro-symbolic technique, NeurTaskSyn, that can synthesize programming tasks for a specification given in the form of desired programming concepts exercised by its solution code and constraints on the visual task. NeurTaskSyn has two components: the first component is trained via imitation learning procedure to generate possible solution codes, and the second component is trained via reinforcement learning procedure to guide an underlying symbolic execution engine that generates visual tasks for these codes. We demonstrate the effectiveness of NeurTaskSyn through an extensive empirical evaluation and a qualitative study on reference tasks taken from the *Hour of Code: Classic Maze* challenge by Code.org and the *Intro to Programming with Karel* course by CodeHS.com.

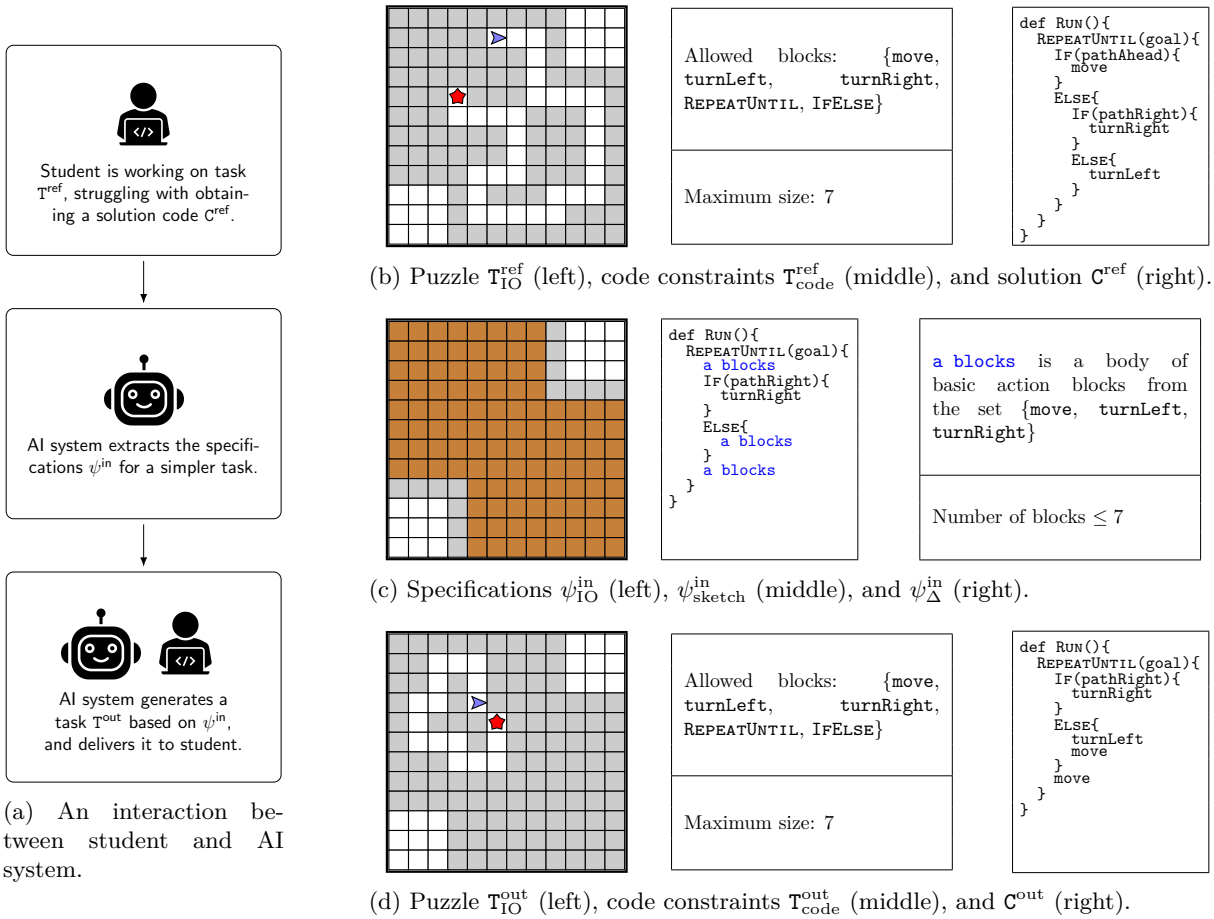
1 Introduction

Recent advances in generative AI have demonstrated impressive performance in a variety of domains, including visual art and music creation (Dong et al., 2018; Briot et al., 2020; Suh et al., 2021; Ramesh et al., 2021; Rombach et al., 2022), medicinal chemistry synthesis (Schneider et al., 2020; Walters & Murcko, 2020; Tong et al., 2021; Gao & Coley, 2020), and AI-enhanced programming (Finnie-Ansley et al., 2022; Leinonen et al., 2023; Chen et al., 2021; Feng et al., 2020; Phung et al., 2023a). These successes are, in part, driven by advanced capabilities of deep generative models, such as Stable Diffusion (Rombach et al., 2022), ChatGPT (OpenAI, 2023a), and GPT-4 (OpenAI, 2023b). These advancements also hold great promise in enhancing education, for instance, by generating personalized content and new practice tasks for students allowing them to master required concepts (Sarsa et al., 2022; Tate et al., 2023; Baidoo-Anu & Owusu Ansah, 2023; Lim et al., 2023; Phung et al., 2023b).

In this paper, we explore the role of generative AI in visual programming domains used for introductory programming. Popular domains, such as Scratch (Resnick et al., 2009), *Hour of Code:Maze Challenge* by Code.org (HoCMaze) (Code.org, 2013b;a), and Karel (Pattis et al., 1995), have become an integral part of introductory computer science education and are used by millions of students (Code.org, 2013a; Wu et al., 2019; Price & Barnes, 2017). In existing visual programming platforms, programming tasks are hand-curated by tutors and the available set of tasks is typically very limited, posing a major hurdle for novices in mastering the missing concepts (Zhi et al., 2019; Ahmed et al., 2020). To this end, we seek to design generative models that can automatically synthesize visual programming tasks for a given specification (see Figure 1).

1.1 Motivation and Overview

Our work is motivated by existing literature on AI-driven educational systems that seek to provide feedback to students who are struggling while solving a programming task. For instance, (Zhi et al., 2019) studies feedback provided in the form of worked examples as demonstration and (Ghosh et al., 2022) studies feedback provided in the form of new simpler tasks for adaptive scaffolding. We have illustrated the interaction between such a system and a student in Figure 1. Previous works have focused on studying how



(a) An interaction between student and AI system.

(b) Puzzle T^{ref}_{IO} (left), code constraints T^{ref}_{code} (middle), and solution C^{ref} (right).

(c) Specifications ψ^{in}_{IO} (left), ψ^{in}_{sketch} (middle), and ψ^{in}_{Δ} (right).

(d) Puzzle T^{out}_{IO} (left), code constraints T^{out}_{code} (middle), and C^{out} (right).

Figure 1: **(a)** Illustration of an AI system helping a student, inspired by (Ghosh et al., 2022). The AI system supports the student when trying to solve T^{ref}. The system first extracts a specification for a simpler task that can be provided to help the student. It then generates a new task based on that specification and delivers it to the student. **(b)** shows reference task T^{ref} along with solution code C^{ref}. **(c)** shows specifications ψ^{in} extracted by AI system. **(d)** shows task T^{out} and code C^{out} synthesized by AI system.

to adapt the feedback type according to the student's needs. We complement these works by focusing on generating on-the-fly the desired examples and simpler tasks that the system could provide to the student. In our work, we consider that a desired feedback format for a student is an input specification in the form of puzzle layout, code structure, code size, and selected programming concepts. Since the search space defined by these input specifications is potentially unbounded, designing AI systems that can synthesize correct tasks in real-time for any input specification is critical to cater to a diverse set of students' behaviors.

As a natural approach, one might be tempted to employ state-of-the-art large language models (LLMs) to generate a visual programming task by providing task synthesis specification as a prompt. In particular, models like GPT-4 are trained on multi-modal data, including text, code, and visual data, and hence it seems a suitable technique for reasoning about visual programming tasks (OpenAI, 2023b; Bubeck et al., 2023). However, our results show that these models are ineffective in synthesizing visual programming tasks and struggle with logical and spatial reasoning, as also indicated in recent literature on state-of-the-art models (Bang et al., 2023; Bubeck et al., 2023; Valmeekam et al., 2022; Huang & Chang, 2022); see Section 5 for detailed discussion. In general, a major challenge in using purely neural generative models for synthesizing visual programming tasks is that the generative process is highly brittle – even a small modification in the output task could make it invalid or semantically incorrect w.r.t. the input specification (Ahmed et al., 2020).

As an alternative to neural generative models, we could rely on symbolic generative methods driven by search and planning algorithms to generate content that matches a specification. Several works have

shown the efficacy of symbolic methods to generate new tasks in various educational domains, e.g., algebra exercises (Singh et al., 2012; Gulwani, 2014), geometric proof problems (Alvin et al., 2014), natural deduction (Ahmed et al., 2013), mathematical word problems (Polozov et al., 2015), Sokoban puzzles (Kartal et al., 2016), and visual programming tasks (Ahmed et al., 2020; Ghosh et al., 2022). In particular, our work is related to (Ahmed et al., 2020; Ghosh et al., 2022) that proposed symbolic methods guided by hand-crafted constraints and Monte Carlo Tree Search to generate high-quality visual programming tasks. However, their symbolic methods still suffer from intractably large spaces of feasible tasks and codes for a given specification, and could take several minutes to generate an output task for an input specification; see Section 4 for comparison to our technique. In general, a major shortcoming of using purely symbolic generative methods in the above-mentioned works is that the generative process is typically time-inefficient and not suitable for applications that require real-time synthesis.

Against that backdrop, the main research question is: *Can we develop neuro-symbolic techniques that can synthesize high-quality visual programming tasks while being robust and efficient?* To this end, we develop NeurTaskSyn, a novel neuro-symbolic technique that can synthesize programming tasks for input specifications in the form of desired programming concepts exercised by its solution code and constraints on the visual task. Given a task synthesis specification as input (Figure 1c), NeurTaskSyn uses two components trained via reinforcement learning procedure: the first component generates possible solution codes, and the second component involves guiding an underlying symbolic execution engine that generates visual tasks for these codes. Our main results and contributions are summarized below: I. We formulate visual programming puzzle synthesis as a sequential decision-making process and propose NeurTaskSyn, a novel neuro-symbolic technique for synthesizing visual programming tasks (Section 3). II. We demonstrate the performance of NeurTaskSyn by comparing it to baselines from existing works (Section 4). III. We demonstrate the effectiveness of NeurTaskSyn through an extensive evaluation on task specifications from real-world programming platforms (Section 5). IV. We will release the implementation and datasets to facilitate future research.

1.2 Related work

Educational task generation. Earlier works on task generation considered simpler domains such as algebra problems where solutions follow well-defined procedures. These works employed classical methods such as template-based (Polozov et al., 2015; Singh et al., 2012) or exhaustive enumeration-based task generation (Ahmed et al., 2020; Alvin et al., 2014). We aim to synthesize programming task given well-structured specifications, similar to previous works on visual task synthesis (Ahmed et al., 2020; Ghosh et al., 2022). While the existing works focused on offline generation, we seek to learn neural models that can generate tasks in real-time, as highlighted in Figure 1. Recent works have also explored the use of LLMs to generate new assignments for Python programming (Sarsa et al., 2022; Phung et al., 2023b) with mixed results.

Spatial and logical reasoning for LLMs. Recent studies have focused on assessing the various capabilities of LLMs (Bubeck et al., 2023; Arora et al., 2023; Bang et al., 2023). These works show that such state-of-the-art models have already achieved impressive generative capabilities for several programming education scenarios, including program repair and hint generation (Sarsa et al., 2022; Leinonen et al., 2023). However, these models still lack crucial capabilities like program execution, symbolic reasoning, and planning that are needed for structured task generation in visual programming domains (Valmeekam et al., 2022; Huang & Chang, 2022; Kaddour et al., 2023; Phung et al., 2023b). Based on (Huang et al., 2023), the LLMs' inherent capabilities alone are still insufficient for many general reasoning tasks and require the guidance of an external tool or a human.

2 Problem Setup

Visual programming tasks. We define a task as a tuple $\mathtt{T} := (\mathtt{T}_{\text{IO}}, \mathtt{T}_{\text{code}})$, where $\mathtt{T}_{\text{IO}}$ denotes the visual puzzle and $\mathtt{T}_{\text{code}}$ denotes additional constraints on a solution code of this puzzle. This task space is general enough to encompass popular visual programming domains, including block-based programming domain of *Hour of Code:Maze Challenge* by Code.org (HoCMaze) (Code.org, 2013b;a) and text-based programming domain of Karel (Pattis et al., 1995). For instance, the task in Figure 1b is based on HoCMaze:Maze20 task Code.org (2013a) – it is comprised of a visual puzzle where a solution code, when executed, will take

the avatar to the goal without crashing into walls. Additional constraints on a solution code specify that it should use only blocks from the set {move, turnLeft, turnRight, REPEATUNTIL, IFELSE} and have a size ≤ 7. We give a similar illustrative example for Karel in Section 5.

Solution codes of a task. We define the space of all possible codes $\mathbb{C}$ in a domain via a domain-specific language (DSL) (Gulwani et al., 2017). For instance, in our evaluation with HoCMaze and Karel programming domains, we will use their corresponding DSLs as introduced in (Bunel et al., 2018; Ahmed et al., 2020). For a given task $\mathtt{T}$, a code $\mathtt{C} \in \mathbb{C}$ is a solution code if the following holds: $\mathtt{C}$ successfully solves $\mathtt{T}_{\text{IO}}$ while respecting $\mathtt{T}_{\text{code}}$. For example, the codes in Figures 1b and 1d use only blocks from the set {move, turnLeft, turnRight, REPEATUNTIL, IFELSE}, have sizes equal to 7, and when executed, will take the avatar to the goal. Note that IFELSE is considered a single block.

Task synthesis specification. We now introduce a notation to specify desired tasks for synthesis that exercise certain programming concepts in their solution codes and respect certain constraints on the visual puzzle. We define a task synthesis specification as a tuple $\psi := (\psi_{\text{IO}}, \psi_{\text{sketch}}, \psi_{\Delta})$, where ψ_{sketch} is a code sketch (i.e., a partial code) capturing the structure that should be followed by the synthesized task's solution codes, ψ_{IO} is a partially initialized visual puzzle, and ψ_{Δ} specify additional constraints on solution codes. For instance, the input task synthesis specification ψ^{in} in Figure 1c is extracted from $\mathtt{T}^{\text{ref}}$ and $\mathtt{C}^{\text{ref}}$ in Figure 1b – here, $\psi^{\text{in}}_{\text{IO}}$ is a 12x12 maze with certain cells initialized to free or wall cells; $\psi^{\text{in}}_{\text{sketch}}$ specifies the structure and some already initialized conditionals, loops and actions that should be used in solution codes; $\psi^{\text{in}}_{\Delta}$ specifies constraints related to what kind of blocks can be used and the size of a solution code.

Synthesis objective. Given a task synthesis specification $\psi^{\text{in}} := (\psi^{\text{in}}_{\text{IO}}, \psi^{\text{in}}_{\text{sketch}}, \psi^{\text{in}}_{\Delta})$ as input, we seek to generate a task $\mathtt{T}^{\text{out}} := (\mathtt{T}^{\text{out}}_{\text{IO}}, \mathtt{T}^{\text{out}}_{\text{code}})$ as output. Inspired by human-centered task quality criteria for the visual programming domains (Ahmed et al., 2020; Ghosh et al., 2022), we design objectives that capture the quality of desirable tasks. To formally set our synthesis objective and evaluation metrics, below we introduce different binary criteria (taking values 1 or 0) that we want $\mathtt{T}^{\text{out}}$ to satisfy w.r.t. ψ^{in}:

- **O1:Validity.** A task $\mathtt{T}^{\text{out}}$ is valid (i.e., value 1) iff: (a) $\mathtt{T}^{\text{out}}_{\text{IO}}$ respects $\psi^{\text{in}}_{\text{IO}}$, (b) $\mathtt{T}^{\text{out}}_{\text{code}}$ respects $(\psi^{\text{in}}_{\text{sketch}}, \psi^{\text{in}}_{\Delta})$, and (c) there exists at least one solution code $\mathtt{C} \in \mathbb{C}$ for $\mathtt{T}^{\text{out}}$.
- **O2:Concepts.** A task $\mathtt{T}^{\text{out}}$ conceptually captures specification ψ^{in} (i.e., value 1) iff: (a) there exists at least one solution code $\mathtt{C} \in \mathbb{C}$ that respects $\psi^{\text{in}}_{\text{sketch}}$ and (b) all solution codes $\mathtt{C} \in \mathbb{C}$ are at least of the complexity required by $\psi^{\text{in}}_{\text{sketch}}$. We define complexity w.r.t. the total number of programming constructs used (e.g., REPEATUNTIL, WHILE, IF, IFELSE) and the level of code nesting. We refer to the level of nesting as code depth in the remaining content of the paper.
- **O3:Trace.** A task $\mathtt{T}^{\text{out}}$ captures the following synthesis property: for any solution $\mathtt{C}$ for $\mathtt{T}^{\text{out}}$ that respects $(\psi^{\text{in}}_{\text{sketch}}, \psi^{\text{in}}_{\Delta})$, the execution trace of $\mathtt{C}$ on $\mathtt{T}^{\text{out}}$ executes each loop or conditional at least n times. This property is inspired by real-world tasks that are easy to comprehend, as indicated by task quality criteria used in (Ahmed et al., 2020; Ghosh et al., 2022); we will use $n = 2$ in Section 5 evaluation.
- **O4:Overall.** This objective is 1 only if all the above objectives (O1, O2, O3) are satisfied for a task $\mathtt{T}^{\text{out}}$.
- **O5:Human.** This objective captures the quality of a task $\mathtt{T}^{\text{out}}$ from an expert's point of view. The assessed criteria are the conceptual correctness w.r.t. the input specification and the visual quality of the task. Based on these criteria, the expert reports an overall binary score.

3 Our Synthesis Technique NeurTaskSyn

In this section, we present NEURTASKSYN, our neuro-symbolic technique to synthesize visual programming tasks ($\mathtt{T}^{\text{out}}$) for an input specification (ψ^{in}).

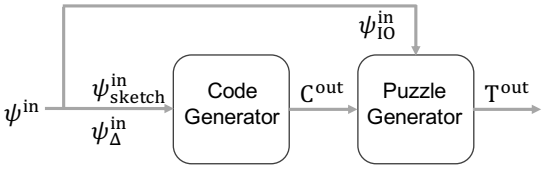


Figure 2: Components of task synthesis.

3.1 Overview

As noted in Section 1, a key challenge in synthesizing tasks is that the mapping from the space of visual tasks

to their solution codes is highly discontinuous – a small modification in the output task could make it invalid or semantically incorrect w.r.t. the input specification (Ahmed et al., 2020). One way to tackle this challenge is to first reason about a possible solution code and then generate visual puzzles based on execution traces of this code (Gulwani, 2014; Kartal et al., 2016; Ahmed et al., 2020; Tercan et al., 2023). This motivates two components in our synthesis process shown in Figure 2: the first component generates possible solution codes $\texttt{C}^{\text{out}}$ (akin to that of program synthesis (Gulwani et al., 2017)); the second component generates the visual puzzle for these codes via symbolic execution (akin to the idea of test-case generation (King, 1976)). After synthesizing both, we can get all the elements of $\texttt{T}^{\text{out}}$. $\texttt{T}^{\text{out}}_{\text{IO}}$ is based on the generated puzzle. $\texttt{T}^{\text{out}}_{\text{code}}$ is extracted from the size and types of blocks of $\texttt{C}^{\text{out}}$.[1] Next, we discuss the components of NeurTaskSyn.

3.2 Generating the Solution Code $\texttt{C}^{\text{out}}$

The code generator component takes the elements of the specification, $(\psi^{\text{in}}_{\text{sketch}}, \psi^{\text{in}}_{\Delta})$, that enforce constraints on solution codes of the desired task and accordingly generates a possible solution code $\texttt{C}^{\text{out}}$. Below, we briefly describe a base symbolic engine to generate syntactically valid codes from specifications via random search and then a neural model to guide this base engine; full details of this component are deferred to Appendix D.1.

Base symbolic engine. The base engine operates on Abstract Syntax Tree representation of code sketches (i.e., partial codes). The engine generates codes by sampling tokens (i.e., action blocks, conditions, and iterators) from underlying DSL while respecting specification. Even though this ensures that a generated code is syntactically correct and valid w.r.t. specification, it could have semantic irregularities. Such irregularities can lead to confusing tasks (e.g., see Figure 8c, where the `If` and `Else` bodies are the same).

Neural model. The neural model is trained to guide the sampling process of the base symbolic engine. This neural model is akin to a program synthesizer, but it has a different objective. It aims to synthesize codes that are semantically correct and can lead to high-quality tasks. One could use a variety of architectures, for instance, transformer-based (Le et al., 2022; Fried et al., 2022; Li et al., 2022; Wang et al., 2021) or custom-made encoder-decoder approaches (Balog et al., 2017; Bunel et al., 2018; Yin & Neubig, 2017). In our work, we use an LSTM-based decoder (Hochreiter & Schmidhuber, 1997) because of their extensive use in the existing literature on generating program solutions for an input visual programming task (Devlin et al., 2017; Bunel et al., 2018; Shin et al., 2019; Gupta et al., 2020). In our setting, the input corresponds to the code specification. Similar to (Bunel et al., 2018), we use an imitation (supervised) learning approach to train the LSTM-based neural model, learning to predict in a token-by-token manner.

3.3 Generating the Visual Puzzle $\texttt{T}^{\text{out}}_{\text{IO}}$

The puzzle generator component takes the element of the specification, $\psi^{\text{in}}_{\text{IO}}$, that enforces constraints on the visual puzzle along with generated code $\texttt{C}^{\text{out}}$ and accordingly generates a visual puzzle $\texttt{T}^{\text{out}}_{\text{IO}}$. We first describe a base symbolic engine that performs symbolic execution of $\texttt{C}^{\text{out}}$ to generate a semantically valid puzzle via random search and then describe a neural model to guide this base engine. We then offer details regarding how we formulate the neurally guided symbolic engine as a reinforcement learning (RL) agent.

Base symbolic engine. The base engine performs symbolic execution of $\texttt{C}^{\text{out}}$ on $\psi^{\text{in}}_{\text{IO}}$ which has uninitialized elements/unknowns. This symbolic execution emulates an execution trace of $\texttt{C}^{\text{out}}$ and makes decisions about unknowns resulting in a concrete instantiation of $\psi^{\text{in}}_{\text{IO}}$ to $\texttt{T}^{\text{out}}_{\text{IO}}$. The outcome of these decisions affects the quality of the generated $\texttt{T}^{\text{out}}_{\text{IO}}$. Figure 3 shows an example of symbolic execution done by the symbolic engine. During this execution process, an emulator will replace unknowns with free cells, walls, the avatar position, and the goal location as indicated by the symbolic engine. The highlighted blocks of code in Figure 3 show the steps where the symbolic execution engine takes decisions, leading to one possible path in the execution of $\texttt{C}^{\text{out}}$ and an instantiation of $\texttt{T}^{\text{out}}_{\text{IO}}$. A code could have a potentially unbounded number of possible execution traces, and randomly taking decisions would typically lead to a lower quality task (King, 1976; Ahmed et al., 2020). Deciding the path of the execution maps to a sequential decision-making process. We aim to guide this decision-making process to achieve execution traces that lead to higher-quality tasks.

[1] It is possible that code $\texttt{C}^{\text{out}}$ generated during intermediate step is not a solution for $\texttt{T}^{\text{out}}$, e.g., when $\texttt{C}^{\text{out}}$ is semantically incorrect and cannot generate a corresponding puzzle. In this case, we set the size constraints in $\texttt{T}^{\text{out}}_{\text{code}}$ as specified by $\psi^{\text{in}}_{\Delta}$.

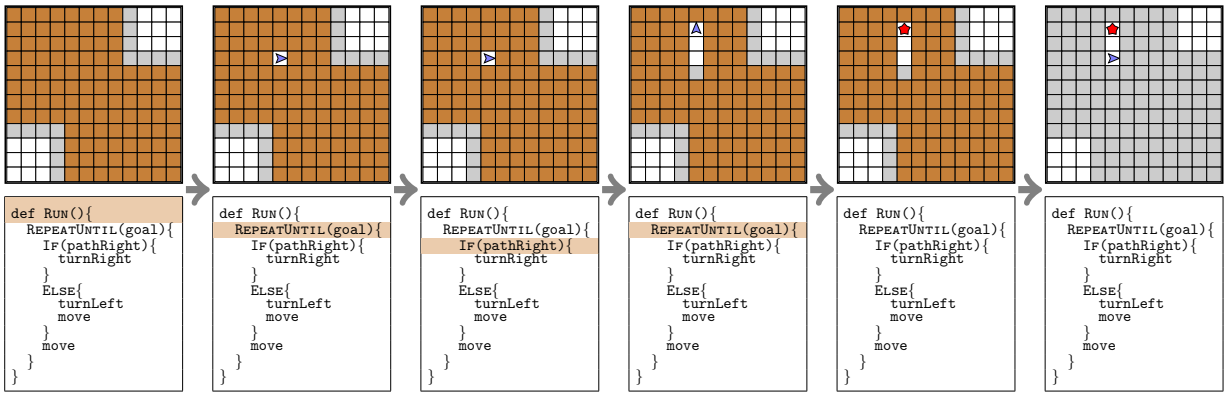


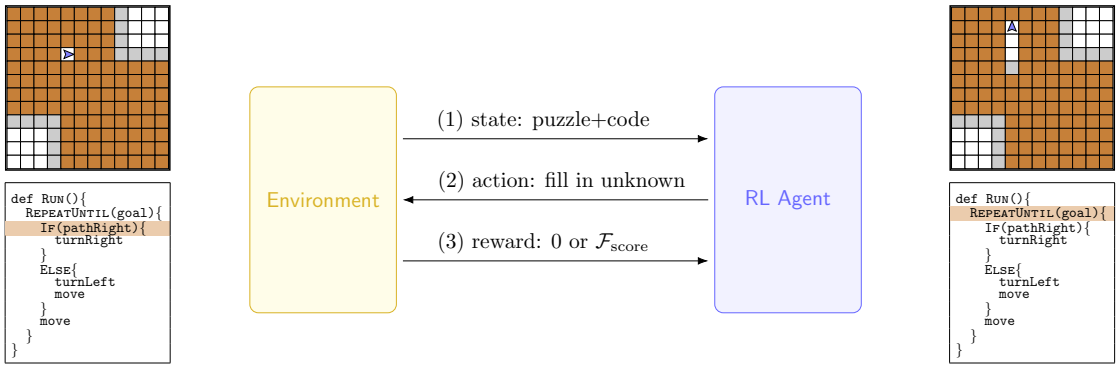
Figure 3: Illustration of symbolic execution. In the first step, the execution of $\mathtt{C}^{\text{out}}$ starts on the partially initialized $\psi^{\text{in}}_{\text{IO}}$. In the second step, the agent's location and orientation are set and execution continues to `RepeatUntil(goal)`, which is decided to `false` in this particular example. This decision does not immediately affect the visual grid. In the third step, the execution continues to `If(pathRight)`, which is decided to `false` and the avatar moves accordingly. The effect of these actions can be seen in the next state of the visual grid. In the fourth step, the execution continues to `RepeatUntil(goal)`, which is decided to `true`, giving the goal location at step five. This leads to the sixth step, which is the end of the execution. The remaining unknowns are replaced with walls.

Neural model. We train the neural model to guide the decision-making process of the base symbolic engine. Our neural architecture uses a CNN-based encoder for partial visual puzzles and combines it with code execution-related features (e.g., coverage, currently executed code block). This choice is based on previous works on program synthesis for visual programming tasks (Bunel et al., 2018; Gupta et al., 2020). Other works have investigated the use of Monte Carlo Tree Search (MCTS) (Kocsis & Szepesvári, 2006) strategy to guide the symbolic execution for generating better puzzles with fewer resources (Kartal et al., 2016; Ahmed et al., 2020). However, these works used MCTS at inference time without any learned policy and could take several minutes to generate an output task for an input specification. Instead, a neural model makes use of learned experience to lead to high-quality puzzles.

Neurally guided symbolic engine as an RL agent. To speed up the generation process at inference, we train an RL agent (Sutton & Barto, 2018) whose reward is defined via a scoring function $\mathcal{F}_{\text{score}}$ that captures the quality of the generated visual puzzle for an input specification; this scoring function is similar in spirit to that used for MCTS in (Kartal et al., 2016; Ahmed et al., 2020). The agent's goal is to make decisions about the unknowns encountered when symbolically executing a code with the objective of generating high-quality puzzles. More concretely, we consider an episodic Markov Decision Process (MDP) defined as a tuple $M = (S, A, P, R, S_0)$, where an episode corresponds to a full symbolic execution of a code and the agent interacts with the environment over discrete time steps t. The elements of the MDP are defined as follows:

- A state $s \in S$ is given by current status of visual puzzle and the block being executed (as in Figure 3);
- A set of state-depended actions A_s can either be the set of initialization actions (i.e., $\{\mathtt{pos}_1, \mathtt{pos}_2, ..., \mathtt{pos}_n\}$), or a set of binary decisions (i.e, $\{\mathtt{true}, \mathtt{false}\}$);
- $P : S \times A \times S \to \mathbb{R}$ denotes deterministic transition dynamics, $P(s'|s, a) = 1$ for $s' = s \oplus a$, and 0 otherwise.
- $R : S \times A \to \mathbb{R}$ denotes a sparse reward. During the episode, the reward values are returned as 0; at the end of an episode, the reward values are computed using $\mathcal{F}_{\text{score}}(\mathtt{T}^{\text{out}}, \mathtt{C}^{\text{out}})$.
- $S_0 \subseteq S$ is the set of initial states. An initial state is composed of $\psi^{\text{in}}_{\text{IO}}$ and $\mathtt{C}^{\text{out}}$.

In order to explain the interaction between the neurally guided symbolic engine and the emulator, we use a concrete example in Figure 4. Let us consider that the current state at time t is composed of the block being executed (i.e., `If(pathRight)` interrogation) and the status of the visual puzzle (i.e., the avatar surrounded



(a) State at time t (b) Interaction at time t (c) State at time $t+1$

Figure 4: Illustration showing the interaction between the RL agent and the environment. **(a)** shows state at time t, composed of code block being executed and status of the visual puzzle. **(b)** shows the interaction between the environment and the RL agent at time t. The environment comprises the code emulator and the scoring function. The RL agent is the neurally guided symbolic engine. **(c)** shows state at time $t+1$

by unknowns). This is depicted in Figure 4a. At this point, the neural model can guide the symbolic engine, as an interaction between the environment and the agent is expected (Figure 4b). The agent receives the current state of the emulator (i.e., as in Figure 4a) and outputs an action, i.e., the decision on how to fill in the unknown. In our case, the decision is that there is *no path to the right*. Then the environment computes a reward in accordance with the action taken by using $\mathcal{F}_{\text{score}}$. After the decision is taken, the emulator continues executing the upcoming blocks (i.e., `turnLeft`, `move`, `move`) until reaching the next step where a decision is needed (i.e., `RepeatUntil(goal)`). The state at time $t+1$ is represented by the block being executed (i.e., `RepeatUntil(goal)`) and the new status of the visual puzzle, as depicted in Figure 4c. The implementation details of the RL agent are provided in Appendix D.2.

4 Training and Validation on Synthetic Specifications

In this section, we train and validate NeurTaskSyn on synthetic datasets of task specifications. We employ different variants of NeurTaskSyn to quantify the utility of individual components and evaluate w.r.t. scoring and runtime metrics. Importantly, the models obtained here through training on synthetic dataset will be used for evaluation on real-world specifications in Section 5.

4.1 Evaluation Setup

Visual programming domains. We consider two popular visual programming domains: *Hour of Code:Maze Challenge* by Code.org (HoCMaze) (Code.org, 2013b;a) and Karel (Pattis et al., 1995), as introduced in Sections 1 and 2. Both these programming domains have been studied extensively in the literature on program/task synthesis (Bunel et al., 2018; Shin et al., 2019; Ahmed et al., 2020; Gupta et al., 2020) and computing education (Piech et al., 2015; Efremov et al., 2020; Ghosh et al., 2022). We define a few domain-specific elements for these visual programming domains. First, as introduced in Section 2, we use two domain-specific languages (DSLs) adapted from (Bunel et al., 2018; Ahmed et al., 2020). Second, as mentioned in Section 3, we will use domain-specific scoring functions $\mathcal{F}_{\text{score}}^{\text{HoCMaze}}$ and $\mathcal{F}_{\text{score}}^{\text{Karel}}$ to capture quality of a visual programming task. In our work, we adapt scoring functions used in (Ahmed et al., 2020). Full details are provided in the supplementary material; in a nutshell, these scoring functions are designed to intuitively capture the synthesis objectives set in Section 2, including properties like code coverage and trace quality. These scoring functions will be used in different ways: (a) during training of NeurTaskSyn's puzzle generator as a reward for RL agent and during inference to select an output task from candidates; (b) when evaluating different techniques with a surrogate metric based on these scoring functions; (c) when computing the quality of expert-designed code-task pairs for the runtime experiments. For each domain, we will make use of an offline, time-intensive, method TaskOracle(`C`): it does one million symbolic executions of a given code `C` and returns highest-scoring task w.r.t. scoring function $\mathcal{F}_{\text{score}}$ (cf. Footnote 2).

Scoring-based metrics. We introduce a binary success metric $\mathcal{M}(\psi^{\text{in}}, \texttt{T}^{\text{out}}, \texttt{C}^{\text{out}})$ that is used to compare the performance of different techniques and to pick hyperparameters. More concretely, $\mathcal{M}(\psi^{\text{in}}, \texttt{T}^{\text{out}}, \texttt{C}^{\text{out}})$ is 1 if the following hold: (i) the task $\texttt{T}^{\text{out}}$ is valid w.r.t. ψ^{in}; (ii) the generated code $\texttt{C}^{\text{out}}$ is semantically correct in a sense that it can lead to a valid task via TaskOracle, i.e., $\mathcal{F}_{\text{score}}(\text{TaskOracle}(\texttt{C}^{\text{out}}), \texttt{C}^{\text{out}}) > \lambda_1$; (iii) the generated task $\texttt{T}^{\text{out}}$ is good quality in comparison to the oracle-generated task, i.e., $\mathcal{F}_{\text{score}}(\texttt{T}^{\text{out}}, \texttt{C}^{\text{out}}) > \lambda_2 \cdot \mathcal{F}_{\text{score}}(\text{TaskOracle}(\texttt{C}^{\text{out}}), \texttt{C}^{\text{out}})$. We use $\lambda_1 = 0$ and $\lambda_2 = 0.9$ in our experiments. For each technique, performance is computed as % success rate across the evaluation dataset w.r.t. $\mathcal{M}$; in total, we compute performance across three seeds and report averaged results as mean (stderr). Importantly, we note that this metric only serves as a surrogate metric for evaluation on synthetic dataset; the neural models trained here will be evaluated on real-world task specifications w.r.t. the synthesis objectives in the next section.

Runtime metrics. We aim to measure the resources needed for a technique to reach certain quality thresholds. We do this by first computing the score given by $\mathcal{F}_{\text{score}}$ to an expert-designed task-code pair. Then, we fix the code and measure the time and number of symbolic execution rollouts needed to reach a given % of the computed score. We evaluate each technique multiple times and report averaged results.

4.2 Evaluation w.r.t. Scoring-based Metrics

Synthetic task specifications. For training and evaluation of techniques, we create a dataset of synthetic task specifications per domain, referred to as $\mathbb{D} := \{\psi^{\text{in}}\}$. To create one specification $\psi^{\text{in}} := (\psi^{\text{in}}_{\text{sketch}}, \psi^{\text{in}}_{\text{IO}}, \psi^{\text{in}}_{\Delta})$, the most crucial part is getting a code sketch $\psi^{\text{in}}_{\text{sketch}}$ that respects the DSL and can lead to a valid code generation. We start by sampling a code $\texttt{C}^{\text{in}}$ from the DSL for a given structure, depth, and constructs – this sampling process is inspired by methods for synthetic dataset creation (Bunel et al., 2018; Shin et al., 2019; Ahmed et al., 2020). For each sampled code, we check its semantic validity, i.e., this code can lead to a high-quality task using TaskOracle. Afterwards, for a sampled code $\texttt{C}^{\text{in}}$ we create its corresponding $\psi^{\text{in}}_{\text{sketch}}$ by keeping only the programming constructs (loops/conditionals) with a random subset of the booleans/iterators masked out. The rest of the ψ^{in} elements are instantiated as follows: $\psi^{\text{in}}_{\text{IO}}$ is 16×16 size without any initialization, $\psi^{\text{in}}_{\Delta}$ enforces the underlying DSL and a randomly initialized size between the number of blocks in $\texttt{C}^{\text{in}}$ and 17. In our evaluation, we split $\mathbb{D}$ as follows: 80% for training the neural models ($\mathbb{D}^{\text{train}}$), 10% for calibration ($\mathbb{D}^{\text{cal}}$), and a fixed 10% for evaluation ($\mathbb{D}^{\text{test}}$).

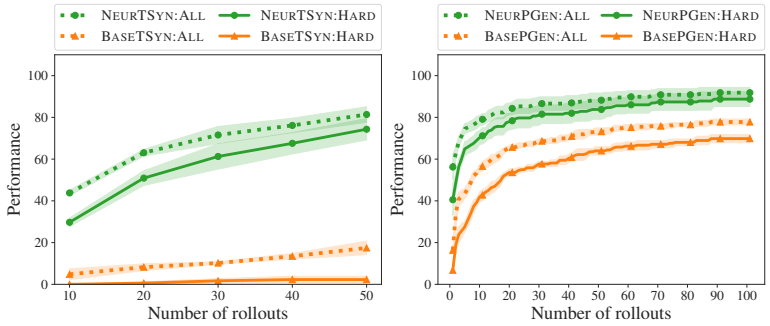
Techniques. First we describe NeurTaskSyn, our main task synthesis technique from Section 3. For each domain (HoCMaze and Karel), we train a separate instance of NeurTaskSyn using the synthetic dataset introduced above. In Section 3, we described the generation process for a single "rollout", i.e., one $\texttt{C}^{\text{out}}$ and one puzzle $\texttt{T}^{\text{out}}_{\text{IO}}$ is generated. In practice, we use multiple rollouts to select a final output task $\texttt{T}^{\text{out}}$. More concretely, at inference time for a given ψ^{in} as input, NeurTaskSyn generation process is captured by two parameters: number of code rollouts c by the code generator and number of puzzle rollouts p by the puzzle generator for each generated code. We denote these hyperparameters in subscript, e.g., $\text{NeurTaskSyn}_{\text{c:5,p:10}}$ for 5×10 rollouts. Out of these $\text{c} \times \text{p}$ candidates, the technique outputs one task $\texttt{T}^{\text{out}}$ along with solution code $\texttt{C}^{\text{out}}$ using its scoring function. Next, we describe different variants of $\text{NeurTaskSyn}_{\text{c,p}}$ and baselines:

- $\text{NeurPuzzleGen}_{\text{c:FIX,p}}$: This technique is a variant of $\text{NeurTaskSyn}_{\text{c,p}}$ to evaluate its puzzle generation component, assuming the code generator has access to code $\texttt{C}^{\text{in}}$ associated with specification ψ^{in} in the dataset. At inference, $\text{NeurPuzzleGen}_{\text{c:FIX,p}}$ generation process is captured by hyperparameter p, i.e., number of puzzle rollouts. Out of p candidates, technique outputs one task, analogous to NeurTaskSyn.

- $\text{BaseTaskSyn}_{\text{c,p}}$ and $\text{BasePuzzleGen}_{\text{c:FIX,p}}$: These techniques operate similar to $\text{NeurTaskSyn}_{\text{c,p}}$ and $\text{NeurPuzzleGen}_{\text{c:FIX,p}}$, but use only symbolic engine with random search.[2]

Results. Figures 5a and 5b report results as we vary the number of rollouts for different techniques, evaluated on the full dataset and on a "hard" segment of the dataset, where $\psi^{\text{in}}_{\text{sketch}}$ uses at least 2 constructs and has a depth of 3. In summary, these results demonstrate the utility of different components of NeurTaskSyn and how the synthesis quality improves as we increase the number of rollouts.

[2] TaskOracle($\texttt{C}$) introduced above is BasePuzzleGen with $\text{p} = 10^6$ rollouts for a fixed code $\texttt{C}$.



(a) BaseTaskSyn, NeurTaskSyn (b) BasePuzzleGen, NeurPuzzleGen

Metric	Rollouts			Runtime
	Quality			Quality
Technique:Data	90%	95%	99%	99%
NeurPGen:**All**	4	9	30	1.97s
NeurPGen:**Hard**	15	46	160	11.08s
BasePGen:**All**	91	2932	69105	1977.44s
BasePGen:**Hard**	437	17122	405231	11800.11s
MCTS:**All**	92	1180	11740	14.92s
MCTS:**Hard**	283	6841	10116	88.25s

(c) Results for runtime evaluation.

Figure 5: **(a)** Results for $\text{BaseTaskSyn}_{c,p}$ and $\text{NeurTaskSyn}_{c,p}$ by increasing code rollouts c from 1 to 5 with fixed puzzle rollouts p = 10. **(b)** Results for $\text{BasePuzzleGen}_{c:\text{FIX},p}$ and $\text{NeurPuzzleGen}_{c:\text{FIX},p}$ by increasing puzzle rollouts p from 1 to 100. **(c)** Table showing the runtime/rollouts comparison between different techniques, on tasks of different difficulty, when varying the required quality threshold. The quality threshold is expressed as % of the quality score of an expert-designed task. See further details in Section 4.

4.3 Evaluation w.r.t. Runtime Metrics

Code samples for puzzle generation. For evaluating the runtime of our technique, we pick a set of codes designed by experts as solutions for six representative tasks from both the HoCMaze and Karel domains. We select those tasks that are common among our evaluation in Section 5 (see Figure 6) and the work of (Ahmed et al., 2020). These tasks are derived from HoC:Maze8, HoC:Maze9, HoC:Maze13, HoC:Maze18 (Code.org, 2013b), and Karel:OurFirst, Karel:Diagonal (CodeHS, 2012b).

Techniques. We focus on evaluating the runtime performance of three techniques. $\text{NeurPuzzleGen}_{c:\text{FIX},p}$ and $\text{BasePuzzleGen}_{c:\text{FIX},p}$ are equivalent to the techniques of Section 4.2. $\text{MCTS}_{c:\text{FIX},p}$ is the puzzle generation method based on Monte Carlo tree search (MCTS) (Ahmed et al., 2020), introduced in Section 3.3. More specifically, we do not set a fixed number of puzzle rollouts p, but we measure the values of p needed to reach certain quality thresholds.

Results. We present our results in Figure 5c, averaged over all the selected tasks. Additionally, we define an expert code as "hard" when using at least 2 constructs and having a depth of 3. We separately report the results on HoC:Maze18, which is considered hard. It can be noticed that the unguided symbolic execution of $\text{BasePuzzleGen}_{c:\text{FIX}}$ leads to large execution times when aiming to synthesize high-quality tasks. $\text{MCTS}_{c:\text{FIX}}$ has faster runtimes than $\text{BasePuzzleGen}_{c:\text{FIX}}$, but it becomes less scalable when required to come up with high-quality tasks for the hard code. $\text{NeurPuzzleGen}_{c:\text{FIX}}$ maintains low execution times and number of rollouts for any of the required quality thresholds, even when faced with a hard code.

5 Experiments on Real-World Specifications

In this section, we evaluate our task synthesis technique NeurTaskSyn on real-world specifications.

Real-world task specifications. We use a set of 10 task specifications from HoCMaze and Karel domains, shown in Figure 6. These task specifications are inspired by their source tasks (see "Source" column) in the following sense: we create a specification ψ^{in} for which the corresponding source task is a desired task as would be created by experts. Figure 8 shows illustration of task synthesis for a variant of $\psi 3$ (source as Maze18 HoCMaze task) where we used 12x12 grid size with certain cells pre-initialized; analogously, Figure 9 shows illustration of task synthesis for a variant of $\psi 8$ (source as Stairway Karel task) where we used 12x12 grid size.

Synthesis quality metrics. We evaluate techniques w.r.t. different metrics, each corresponding to a synthesis objective introduced in Section 2. Even though objectives O1–O4 are quantitative, it is challenging to fully automate their evaluation because it requires analyzing properties of different possible solution codes

ψ^{in}	$\psi^{\text{in}}_{\text{sketch}}$ structure	(depth, constructs)	$\psi^{\text{in}}_{\text{IO}}$	$\psi^{\text{in}}_{\Delta}$	Source
$\psi 0$	{RUN {REPEAT}}	$(2, 1)$	16x16 empty	HoCMaze, blocks ≤ 10	HoC:Maze9 Code.org (2013b)
$\psi 1$	{RUN {REPEATUNTIL}}	$(2, 1)$	16x16 empty	HoCMaze, blocks ≤ 10	HoC:Maze13 Code.org (2013b)
$\psi 2$	{RUN {REPEAT; REPEAT}}	$(2, 2)$	16x16 empty	HoCMaze, blocks ≤ 10	HoC:Maze8 Code.org (2013b)
$\psi 3$	{RUN {REPEATUNTIL{IFELSE}}}	$(3, 2)$	16x16 empty	HoCMaze, blocks ≤ 10	HoC:Maze18 Code.org (2013b)
$\psi 4$	{RUN {REPEATUNTIL{IF; IF}}}	$(3, 3)$	16x16 empty	HoCMaze, blocks ≤ 10	HoC:Maze20 Code.org (2013b)
$\psi 5$	{RUN}	$(1, 0)$	16x16 empty	Karel, blocks ≤ 10	Karel:OurFirst CodeHS (2012b)
$\psi 6$	{RUN {WHILE}}	$(2, 1)$	16x16 empty	Karel, blocks ≤ 10	Karel:Diagonal CodeHS (2012b)
$\psi 7$	{RUN {WHILE; WHILE}}	$(2, 2)$	16x16 empty	Karel, blocks ≤ 10	Karel:RowBack CodeHS (2012b)
$\psi 8$	{RUN {WHILE{IF}}}	$(3, 2)$	16x16 empty	Karel, blocks ≤ 10	Karel:Stairway CodeHS (2012b)
$\psi 9$	{RUN {WHILE{REPEAT}}}	$(3, 2)$	16x16 empty	Karel, blocks ≤ 10	Karel:CleanAll

Figure 6: Real-world task specifications for HoCMaze and Karel; $\psi^{\text{in}}_{\text{sketch}}$ is shortened for brevity; $\psi^{\text{in}}_{\Delta}$ indicated as booleans and actions allowed by each domain, with the addition of maximum size.

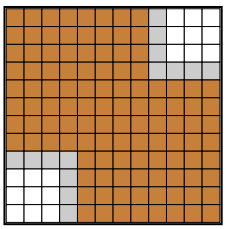
Technique	**O1:Validity**	**O2:Concepts**	**O3:Trace**	**O4:Overall**	**O5:Human**	$\texttt{C}^{\text{out}}$ **solves** $\texttt{T}^{\text{out}}$
NEURTASKSYN$_{\text{c:10,p:100}}$	1.00 (0.00)	0.83 (0.04)	0.80 (0.00)	0.80 (0.00)	0.77 (0.11)	1.00 (0.00)
BASETASKSYN$_{\text{c:10,p:100}}$	0.97 (0.04)	0.37 (0.08)	0.33 (0.04)	0.33 (0.04)	0.20 (0.07)	0.50 (0.12)
GPT4TASKSYN-*converse*	0.97 (0.04)	0.57 (0.11)	0.60 (0.07)	0.43 (0.11)	0.27 (0.08)	0.33 (0.08)
GPT4TASKSYN-*fewshot*	0.80 (0.07)	0.37 (0.11)	0.57 (0.11)	0.33 (0.08)	0.13 (0.11)	0.27 (0.04)
EXPERT	1.00	1.00	1.00	1.00	1.00	1.00

Figure 7: Results on real-world task specifications for HoCMaze and Karel in Figure 6; see Section 5. In this figure, EXPERT refers to expert-designed tasks.

of a generated task. We manually did this evaluation when computing performance for each technique and metric. To capture objective O5, we assess the quality of synthesized tasks using human experts' annotations. Each expert independently evaluated all specifications and all techniques. Additionally, we report a binary metric of whether $\texttt{C}^{\text{out}}$ solves $\texttt{T}^{\text{out}}$ to provide insight into the robustness of the synthesis process. Results are reported as a mean over 10 specifications ψ^{in} from Figure 6; we evaluate over three seeds and report averaged results as mean (stderr). In total, three experts (i.e., one per seed) with experience in visual programming provided annotations for O5.

Techniques. We evaluate NEURTASKSYN$_{\text{c:10,p:100}}$ with c = 10 and p = 100, i.e., total of c × p = 1000 rollouts (see Section 4). Next, we describe additional techniques evaluated:

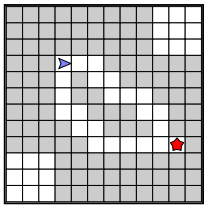
- BASETASKSYN$_{\text{c:10,p:100}}$ operates similarly to NEURTASKSYN$_{\text{c:10,p:100}}$, but uses only base symbolic engine with random search without any neural guidance (see Section 4).
- GPT4TASKSYN-*converse* is based on OpenAI's GPT-4 (OpenAI, 2023b). It uses conversation-style prompts that involved human guidance to correct mistakes. More concretely, it is based on a two-stage process as shown in Figure 2 – we first ask GPT-4 to generate a code $\texttt{C}^{\text{out}}$ for ψ^{in} and then ask it to generate a puzzle $\texttt{T}^{\text{out}}_{\text{IO}}$ that could be solved by $\texttt{C}^{\text{out}}$. The first stage comprised 5 separate queries to generate a $\texttt{C}^{\text{out}}$: each query started with an initial prompt and then follow-up prompts to fix any mistakes. The second stage comprised of another 5 separate queries to generate a puzzle $\texttt{T}^{\text{out}}_{\text{IO}}$: each query started with an initial prompt and then follow-up prompts to fix any mistakes. Once we get $\texttt{C}^{\text{out}}$ and $\texttt{T}^{\text{out}}_{\text{IO}}$, we set $\texttt{T}^{\text{out}}_{\text{code}}$ as for NEURTASKSYN. We defer the prompts to the supplementary material. This variant of GPT4TASKSYN was used to synthesize the tasks in Figures 8 and 9.
- GPT4TASKSYN-*fewshot* uses the same two-stage synthesis process, but employs few-shot examples without any follow-up conversations.
- EXPERT refers to expert-designed tasks. In our setup, EXPERT simply outputs a task $\texttt{T}^{\text{out}}$ based on the source task associated with input specification ψ^{in}; moreover it appropriately adjusts $\texttt{T}^{\text{out}}_{\text{IO}}$ to match $\psi^{\text{in}}_{\text{IO}}$ layout and sets $\texttt{T}^{\text{out}}_{\text{code}}$ as allowed by $(\psi^{\text{in}}_{\text{sketch}}, \psi^{\text{in}}_{\Delta})$ and the size of the minimal solution code.



```
def Run(){
  a blocks
  RepeatUntil(goal){
    a blocks
    If(b){
      a blocks
    }
    Else{
      a blocks
    }
    a blocks
  }
}
```

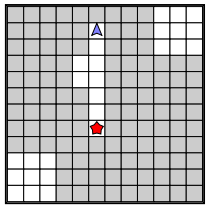
a blocks is a body of basic action blocks from the set {move, turnLeft, turnRight}
b is a boolean condition from {pathAhead, pathLeft, pathRight}
Number of blocks should be ≤ 7

(a) Task synthesis specification ψ^{in}, with $\psi^{\text{in}}_{\text{IO}}$ (left), $\psi^{\text{in}}_{\text{sketch}}$ (middle) and $\psi^{\text{in}}_{\Delta}$ (right).



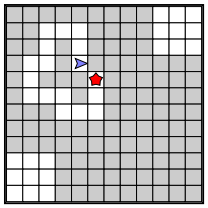
```
def Run(){
  RepeatUntil(goal){
    If(pathLeft){
      turnLeft
      move
    }
    Else{
      move
    }
  }
}
```

(b) $\mathtt{T}^{\text{out}}_{\text{IO}}$, $\mathtt{C}^{\text{out}}$ by GPT4TaskSyn



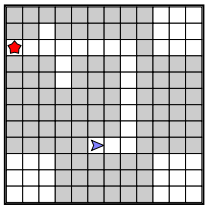
```
def Run(){
  turnLeft
  turnLeft
  RepeatUntil(goal){
    If(pathRight){
      move
    }
    Else{
      move
    }
  }
}
```

(c) $\mathtt{T}^{\text{out}}_{\text{IO}}$, $\mathtt{C}^{\text{out}}$ by BaseTaskSyn



```
def Run(){
  RepeatUntil(goal){
    If(pathRight){
      turnRight
    }
    Else{
      turnLeft
      move
    }
    move
  }
}
```

(d) $\mathtt{T}^{\text{out}}_{\text{IO}}$, $\mathtt{C}^{\text{out}}$ by NeurTaskSyn



```
def Run(){
  RepeatUntil(goal){
    If(pathAhead){
      move
    }
    Else{
      turnLeft
    }
  }
}
```

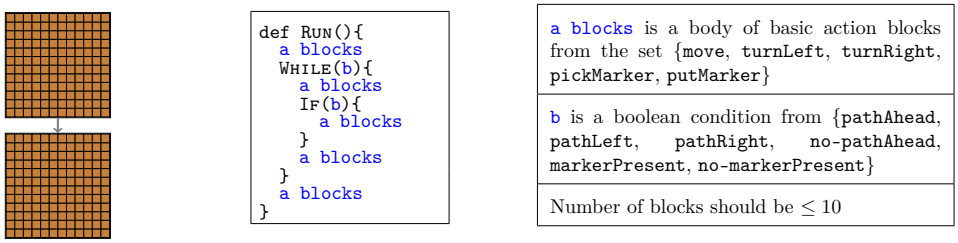
(e) $\mathtt{T}^{\text{out}}_{\text{IO}}$, $\mathtt{C}^{\text{out}}$ by Expert

Figure 8: Illustrative example showcasing task synthesis inspired by the Maze18 HoCMaze task Code.org (2013b;a). The given specification seeks to synthesize tasks where a solution code has the {RepeatUntil{IfElse}} structure. **(a)** Task synthesis specification $\psi^{\text{in}} := (\psi^{\text{in}}_{\text{sketch}}, \psi^{\text{in}}_{\text{IO}}, \psi^{\text{in}}_{\Delta})$ is provided as input: $\psi^{\text{in}}_{\text{IO}}$ is a 12x12 maze with certain cells initialized to free (white) or wall (gray) cells, the rest being marked as unknowns (brown); $\psi^{\text{in}}_{\Delta}$ specifies constraints on code solutions of a synthesized task. **(b–d)** show tasks $\mathtt{T}^{\text{out}}$ and codes $\mathtt{C}^{\text{out}}$ used as intermediate step to generate output tasks by three techniques. **(e)** shows task $\mathtt{T}^{\text{out}}$ and code $\mathtt{C}^{\text{out}}$ based on See Section 5.

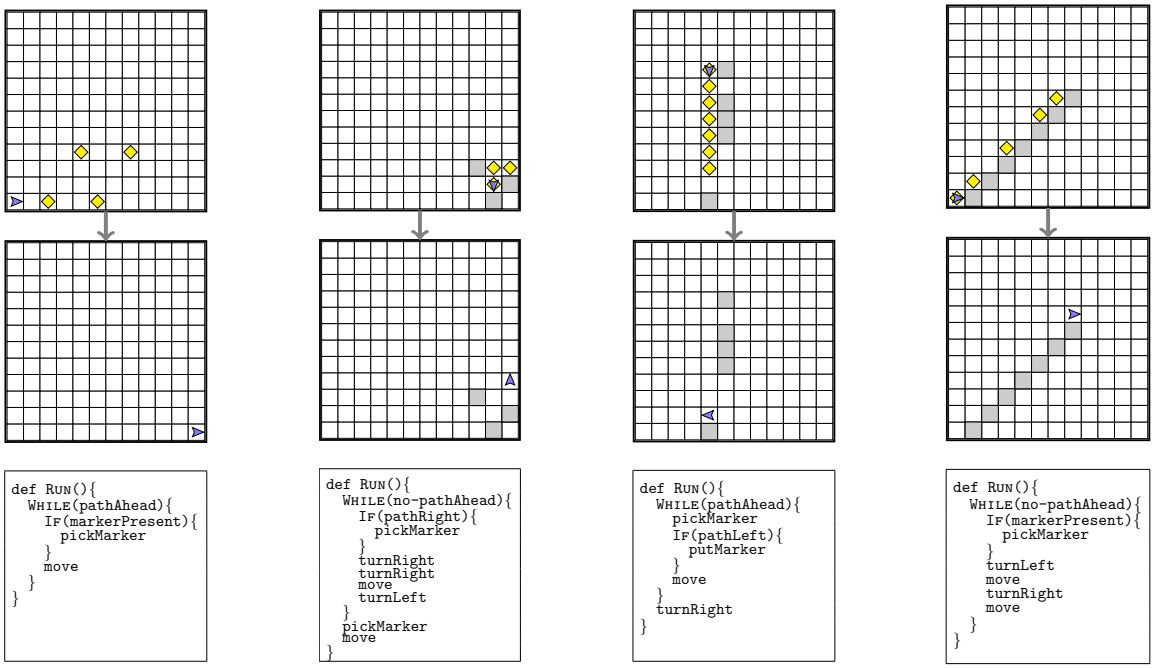
Results. Figure 7 reports evaluation results for different techniques w.r.t. our task synthesis objectives. Next, we summarize some of our key findings. First, NeurTaskSyn has high performance of at least 0.77 across all metrics. The illustrative examples in Figures 8 and 9 showcase the high-quality of tasks synthesized by NeurTaskSyn, matching interesting characteristics of real-world tasks from Expert. Second, BaseTaskSyn, GPT4TaskSyn-*converse*, and GPT4TaskSyn-*fewshot* struggle on objectives O2 and O3. Their low performance can be explained, in part, by failure to generate a valid task/code pair. The illustrative examples in Figures 8 and 9 further highlight the issues of tasks generated by these techniques. For instance, BaseTaskSyn's $\mathtt{T}^{\text{out}}$ in Figure 8c can be solved by a simpler code with lower depth than specified in the input specification; GPT4TaskSyn-*converse*'s $\mathtt{T}^{\text{out}}$ in Figures 8b and 9b are not solvable by codes that would match the input specification. GPT4TaskSyn-*fewshot* shows that changing the prompting strategy does not help with increasing the performance. In summary, these results demonstrate the effectiveness of NeurTaskSyn in synthesizing high-quality visual programming tasks for real-world specifications and some of the challenges in synthesizing visual programming tasks by state-of-the-art neural generative models as the synthesis process requires logical, spatial, and programming skills.

6 Concluding Discussions

We developed a novel neuro-symbolic technique, NeurTaskSyn, that can synthesize visual programming tasks for a given specification. We demonstrated both the efficiency and effectiveness of NeurTaskSyn through an extensive evaluation on synthetically generated and reference tasks picked from popular visual programming environments. We believe our proposed technique has the potential to enhance introductory programming education by synthesizing personalized content. Moreover, we showcase the challenges that



(a) Task synthesis specification ψ^{in}, with $\psi^{\text{in}}_{\text{IO}}$ (left), $\psi^{\text{in}}_{\text{sketch}}$ (middle) and $\psi^{\text{in}}_{\Delta}$ (right).



(b) $\mathtt{T}^{\text{out}}_{\text{IO}}$, $\mathtt{C}^{\text{out}}$ by GPT4TaskSyn (c) $\mathtt{T}^{\text{out}}_{\text{IO}}$, $\mathtt{C}^{\text{out}}$ by BaseTaskSyn (d) $\mathtt{T}^{\text{out}}_{\text{IO}}$, $\mathtt{C}^{\text{out}}$ by NeurTaskSyn (e) $\mathtt{T}^{\text{out}}_{\text{IO}}$, $\mathtt{C}^{\text{out}}$ by Expert

Figure 9: Illustrative example showcasing task synthesis inspired by the Stairway Karel task Pattis et al. (1995); CodeHS (2012b;a). This specification seeks to synthesize tasks where a solution code has the {While{If}} structure. **(a)** Task synthesis specification $\psi^{\text{in}} := (\psi^{\text{in}}_{\text{sketch}}, \psi^{\text{in}}_{\text{IO}}, \psi^{\text{in}}_{\Delta})$ is provided as input: $\psi^{\text{in}}_{\text{IO}}$ is a pregrid-postgrid pair with unitialized grid of size 12x12 ; $\psi^{\text{in}}_{\text{sketch}}$ along with $\psi^{\text{in}}_{\Delta}$ specify constraints on code solutions of a synthesized task. **(b–d)** show tasks $\mathtt{T}^{\text{out}}$ and codes $\mathtt{C}^{\text{out}}$ used as intermediate step to generate output tasks by three techniques. **(e)** shows task $\mathtt{T}^{\text{out}}$ and code $\mathtt{C}^{\text{out}}$ based on Stairway. See Section 5.

purely neural generative models and purely symbolic models face. BaseTaskSyn, a purely symbolic model, can lead to semantic irregularities in codes and low-quality tasks. On the other hand, purely generative techniques based on GPT-4 struggle to generate high-quality puzzles. This result aligns with findings in contemporary studies that GPT-4 may face challenges in code execution, symbolic operations, test-case generation, and task synthesis (Bubeck et al., 2023; Phung et al., 2023b).

There are many interesting directions for future work. First, we have built our LSTM/CNN-based architecture specialized for task synthesis objectives; it would be interesting to fine-tune models like GPT-4 to improve its capabilities for synthesizing visual programming tasks. Moreover, we can analyze the logical and spatial abilities of these fine-tuned models by curating benchmarks in visual programming domains. Second, our methodology focused on visual programming; it would be interesting to develop generative models for synthesizing tasks in other programming domains, such as synthesizing Python problems that match a certain input-output configuration and other specifications regarding the possible solution codes. Third, our evaluation study considered various objectives capturing human-centered aspects along with expert annotations; in the future, it would also be useful to conduct studies with human learners to evaluate the quality in terms of perceived difficulty or interpretability of synthesized tasks.

References

Umair Z. Ahmed, Sumit Gulwani, and Amey Karkare. Automatically Generating Problems and Solutions for Natural deduction. In *IJCAI*, 2013.

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.

Chris Alvin, Sumit Gulwani, Rupak Majumdar, and Supratik Mukhopadhyay. Synthesis of Geometry Proof Problems. In *AAAI*, 2014.

Daman Arora, Himanshu Gaurav Singh, and Mausam. Have LLMs Advanced Enough? A Challenging Problem Solving Benchmark For Large Language Models. *CoRR*, abs/2305.15074, 2023.

David Baidoo-Anu and Leticia Owusu Ansah. Education in the Era of Generative Artificial Intelligence (AI): Understanding the Potential Benefits of ChatGPT in Promoting Teaching and Learning. *Available at SSRN 4337484*, 2023.

Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. DeepCoder: Learning to Write Programs. In *ICLR*, 2017.

Yejin Bang, Samuel Cahyawijaya, Nayeon Lee, Wenliang Dai, Dan Su, Bryan Wilie, Holy Lovenia, Ziwei Ji, Tiezheng Yu, Willy Chung, Quyet V. Do, Yan Xu, and Pascale Fung. A Multitask, Multilingual, Multimodal Evaluation of Chatgpt on Reasoning, Hallucination, and Interactivity. *CoRR*, abs/2302-04023, 2023.

Jean-Pierre Briot, Gaëtan Hadjeres, and François-David Pachet. *Deep Learning Techniques for Music Generation*. Springer, 2020.

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. Sparks of Artificial General Intelligence: Early Experiments with GPT-4. *CoRR*, abs/2303.12712, 2023.

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

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

CodeHS. CodeHS.com: Teaching Coding and Computer Science. `https://codehs.com/`, 2012a.

CodeHS. Intro to Programming with Karel the Dog. `https://codehs.com/info/curriculum/introkarel`, 2012b.

Code.org. Code.org: Learn Computer Science. `https://code.org/`, 2013a.

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

Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. RobustFill: Neural Program Learning under Noisy I/O. In *ICML*, 2017.

Hao-Wen Dong, Wen-Yi Hsiao, Li-Chia Yang, and Yi-Hsuan Yang. Musegan: Multi-track Sequential Generative Adversarial Networks for Symbolic Music Generation and Accompaniment. In *AAAI*, 2018.

Aleksandr Efremov, Ahana Ghosh, and Adish Singla. Zero-shot Learning of Hint Policy via Reinforcement Learning and Program Synthesis. In *EDM*, 2020.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A Pre-Trained Model for Programming and Natural Languages. In *EMNLP*, 2020.

James Finnie-Ansley, Paul Denny, Brett A. Becker, Andrew Luxton-Reilly, and James Prather. The Robots Are Coming: Exploring the Implications of OpenAI Codex on Introductory Programming. In *ACE*, 2022.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. Incoder: A Generative model for Code Infilling and Synthesis. *CoRR*, abs/2204.05999, 2022.

Wenhao Gao and Connor W Coley. The Synthesizability of Molecules Proposed by Generative Models. *Journal of Chemical Information and Modeling*, 60(12):5714–5723, 2020.

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

Sumit Gulwani. Example-based Learning in Computer-aided STEM Education. *Communications of the ACM*, 57(8):70–80, 2014.

Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. Program Synthesis. *Foundations and Trends® in Programming Languages*, 2017.

Kavi Gupta, Peter Ebert Christensen, Xinyun Chen, and Dawn Song. Synthesize, Execute and Debug: Learning to Repair for Neural Program Synthesis. In *NeurIPS*, 2020.

Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9, 1997.

Jie Huang and Kevin Chen-Chuan Chang. Towards Reasoning in Large Language Models: A Survey. *CoRR*, abs/2212-10403, 2022.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. Large Language Models Cannot Self-Correct Reasoning Yet. *CoRR*, abs/2310.01798, 2023.

Jean Kaddour, Joshua Harris, Maximilian Mozes, Herbie Bradley, Roberta Raileanu, and Robert McHardy. Challenges and Applications of Large Language Models. *CoRR*, abs/2307.10169, 2023.

Bilal Kartal, Nick Sohre, and Stephen J. Guy. Data Driven Sokoban Puzzle Generation with Monte Carlo Tree Search. In *AIIDE*, 2016.

James C. King. Symbolic Execution and Program Testing. *Communications of ACM*, 19(7):385–394, 1976.

Levente Kocsis and Csaba Szepesvári. Bandit based Monte-Carlo Planning. In *ECML*, 2006.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu-Hong Hoi. Coderl: Mastering Code Generation through Pretrained Models and Deep Reinforcement Learning. In *NeurIPS*, 2022.

Juho Leinonen, Arto Hellas, Sami Sarsa, Brent N. Reeves, Paul Denny, James Prather, and Brett A. Becker. Using Large Language Models to Enhance Programming Error Messages. In *SIGCSE*, 2023.

Yujia Li et al. Competition-Level Code Generation with Alphacode. *CoRR*, abs/2203.07814, 2022.

Weng Marc Lim, Asanka Gunasekara, Jessica Leigh Pallant, Jason Ian Pallant, and Ekaterina Pechenkina. Generative AI and the Future of Education: Ragnarök or Reformation? A Paradoxical Perspective from Management Educators. *The International Journal of Management Education*, 21(2):100790, 2023.

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

OpenAI. GPT-4 Technical Report. *CoRR*, abs/2303.08774, 2023b.

Richard E Pattis, Jim Roberts, and Mark Stehlik. *Karel the Robot: A Gentle Introduction to the Art of Programming*. John Wiley & Sons, Inc., 1995.

Tung Phung, José Cambronero, Sumit Gulwani, Tobias Kohn, Rupak Majumdar, Adish Singla, and Gustavo Soares. Generating high-precision feedback for programming syntax errors using large language models. In *EDM*, 2023a.

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

Chris Piech, Mehran Sahami, Jonathan Huang, and Leonidas J. Guibas. Autonomously Generating Hints by Inferring Problem Solving Policies. In *L@S*, 2015.

Oleksandr Polozov, Eleanor O'Rourke, Adam M. Smith, Luke Zettlemoyer, Sumit Gulwani, and Zoran Popovic. Personalized Mathematical Word Problem Generation. In *IJCAI*, 2015.

Thomas W. Price and Tiffany Barnes. Position Paper: Block-Based Programming Should Offer Intelligent Support for Learners. *IEEE Blocks and Beyond Workshop*, 2017.

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

Mitchel Resnick, John H. Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay S. Silver, Brian Silverman, and Yasmin B. Kafai. Scratch: Programming for All. *Communications of ACM*, 52(11):60–67, 2009.

Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-Resolution Image Synthesis with Latent Diffusion Models. In *CVPR*, 2022.

Sami Sarsa, Paul Denny, Arto Hellas, and Juho Leinonen. Automatic Generation of Programming Exercises and Code Explanations Using Large Language Models. In *ICER*, 2022.

Petra Schneider, W Patrick Walters, Alleyn T Plowright, Norman Sieroka, Jennifer Listgarten, Robert A Goodnow Jr, Jasmin Fisher, Johanna M Jansen, José S Duca, Thomas S Rush, et al. Rethinking Drug Design in the Artificial Intelligence Era. *Nature Reviews Drug Discovery*, 19(5):353–364, 2020.

Richard Shin, Neel Kant, Kavi Gupta, Chris Bender, Brandon Trabucco, Rishabh Singh, and Dawn Song. Synthetic Datasets for Neural Program Synthesis. In *ICLR*, 2019.

Rohit Singh, Sumit Gulwani, and Sriram K. Rajamani. Automatically Generating Algebra Problems. In *AAAI*, 2012.

Minhyang Suh, Emily Youngblom, Michael Terry, and Carrie J. Cai. AI as Social Glue: Uncovering the Roles of Deep Generative AI during Social Music Composition. In *CHI*, 2021.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT press, 2018.

Tamara Tate, Shayan Doroudi, Daniel Ritchie, and Ying Xu. Educational Research and AI-Generated Writing: Confronting the Coming Tsunami. 2023.

Alperen Tercan, Ahana Ghosh, Hasan Ferit Eniser, Maria Christakis, and Adish Singla. Synthesizing a Progression of Subtasks for Block-Based Visual Programming Tasks. *CoRR*, abs/2305.17518, 2023.

Xiaochu Tong, Xiaohong Liu, Xiaoqin Tan, Xutong Li, Jiaxin Jiang, Zhaoping Xiong, Tingyang Xu, Hualiang Jiang, Nan Qiao, and Mingyue Zheng. Generative Models for De Novo Drug Design. *Journal of Medicinal Chemistry*, 64(19):14011–14027, 2021.

Karthik Valmeekam, Alberto Olmo Hernandez, Sarath Sreedharan, and Subbarao Kambhampati. Large Language Models Still Can't Plan (A Benchmark for LLMs on Planning and Reasoning about Change). *CoRR*, abs/2206-10498, 2022.

W. Walters and Mark Murcko. Assessing the Impact of Generative AI on Medicinal Chemistry. *Nature Biotechnology*, 38(2):143–145, 2020.

Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *EMNLP*, 2021.

Mike Wu, Milan Mosse, Noah D. Goodman, and Chris Piech. Zero Shot Learning for Code Education: Rubric Sampling with Deep Learning Inference. In *AAAI*, 2019.

Pengcheng Yin and Graham Neubig. A Syntactic Neural Model for General-Purpose Code Generation. In *ACL*, 2017.

Rui Zhi, Thomas W. Price, Samiha Marwan, Alexandra Milliken, Tiffany Barnes, and Min Chi. Exploring the Impact of Worked Examples in a Novice Programming Environment. In *SIGCSE*, 2019.

A Table of Contents

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

- Appendix B provides a discussion of the broader impact of our work and compute resources used.
- Appendix C presents the details about the generation of the illustrative examples from Figures 8 and 9, and shows their relationship with metrics O1-O4 described in Section 2.
- Appendix D gives more insights into the architecture described in Section 3.
- Appendix E provides additional details about results, the scoring functions, the synthetic dataset creation process, and the training process in Section 4.
- Appendix F provides the source task/code pairs used for creating the real-world task specifications in Section 5. It also provides more insights into the interaction with GPT-4.

B Discussion

Broader impact. This paper develops new techniques which have the potential of being used for improving pedagogy in visual programming environments. On the existing platforms, content is hand-curated by tutors, offering limited resources for students to practice on. We aim to tackle this challenge by synthesizing novel practice tasks that match a desired level of difficulty with regard to exercised content for a student. We believe our proposed technique has the potential to drastically enhance introductory programming education by synthesizing personalized content for students.

Compute resources. All the experiments were conducted on a cluster of machines equipped with Intel Xeon Gold 6142 CPUs running at a frequency of 2.60GHz.

C Illustrative Examples: Details

In this section, we discuss the details regarding the generation and scoring for each of the techniques' output in Figure 8 for HoCMaze and Figure 9 for Karel.

C.1 Example for HoCMaze in Figure 8

We present $\texttt{T}^{\text{out}}$, along with $\texttt{C}^{\text{out}}$ for each of the techniques with ψ^{in} as input in Figure 11; this figure expands on Figure 8 with additional details. We give additional explanations regarding how each of the techniques' output respects or not metrics O1-O3 (see Sections 2 and 5) in Figure 10.

Generation/adjustment for GPT4TaskSyn in Figure 8. When querying GPT-4 for this example, we set $\psi^{\text{in}}_{\text{IO}}$ as an empty 8x8 grid. We then expand the generated grid to a 12x12 grid and manually integrate the pattern seen in Figure 11a to match the specification. We discuss the results obtained by GPT4TaskSyn-*converse*.

Generation/adjustment for BaseTaskSyn and NeurTaskSyn in Figure 8. The neural model for puzzle generation is trained on 16x16 grids, yet the symbolic engine can support the existence of pre-initialized grids. Thus, we mask the upper-left part of the grid (4 rows and 4 columns), obtaining the 12x12 workspace for the technique. On top of that, on the remaining 12x12 grid, we pre-initialize the pattern seen in Figure 11a (i.e., lower-left and upper-right bounded squares).

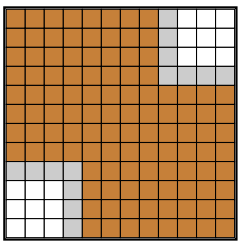
Generation/adjustment for Expert in Figure 8. The output of Expert represents a manual adaptation of the HoC:Maze18 task to expand it to a 12x12 grid and to integrate the pattern seen in in Figure 8a.

Technique	**O1:Validity**	**O2:Concepts**	**O3:Trace**	**O4:Overall**	$\texttt{C}^{\text{out}}$ **solves** $\texttt{T}^{\text{out}}$
GPT4TaskSyn	1	0	0	0	0
BaseTaskSyn	1	0	0	0	1
NeurTaskSyn	1	1	1	1	1
Expert	1	1	1	1	1

Figure 10: Scores showing whether the output $\texttt{T}^{\text{out}}$ for ψ^{in} of each technique respects the four metrics O1-O4, with the additional $\texttt{C}^{\text{out}}$ solves $\texttt{T}^{\text{out}}$ metric, for this HoCMaze example.

We provide explanations for each 0 entry for the objectives O1-O3 in Figure 10:

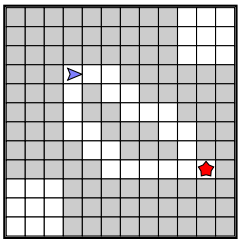
- GPT4TaskSyn for Objective O2: The only possible solution code `C` has depth 4 and uses 3 constructs (nested `IfElse` is needed), i.e., $\psi^{\text{in}}_{\text{sketch}}$ is not respected, hence O2 is 0.
- GPT4TaskSyn for Objective O3: There is no solution `C` that respects $\psi^{\text{in}}_{\text{sketch}}$, hence O3 is 0 by definition.
- GPT4TaskSyn for $\texttt{C}^{\text{out}}$ solves $\texttt{T}^{\text{out}}$: $\texttt{C}^{\text{out}}$, when executed on $\texttt{T}^{\text{out}}_{\text{IO}}$, makes the avatar crash into a wall, hence $\texttt{C}^{\text{out}}$ is not a solution for $\texttt{T}^{\text{out}}$
- BaseTaskSyn for Objective O2: The `IfElse` block employed by $\texttt{C}^{\text{out}}$ is not required. This implies that there is a solution code `C` which has a lower complexity, i.e., a smaller depth and less constructs than required by $\psi^{\text{in}}_{\text{sketch}}$, hence O2 is 0.
- BaseTaskSyn for Objective O3: As the employed `IfElse` block is not required, it is possible to design a solution code that uses `IfElse` with a different conditional (e.g., `If(pathLeft)Else`) for which the body would never be executed, hence O3 is 0.



```
def Run(){
  a blocks
  RepeatUntil(goal){
    a blocks
    If(b){
      a blocks
    }
    Else{
      a blocks
    }
    a blocks
  }
}
```

a blocks is a body of basic action blocks from the set {move, turnLeft, turnRight}
b is a boolean condition from {pathAhead, pathLeft, pathRight}
Number of blocks should be ≤ 7

(a) Task synthesis specification ψ^{in}

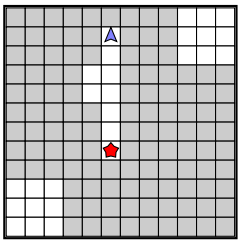


Allowed blocks: {move, turnLeft, turnRight, RepeatUntil, IfElse}
Maximum size: 7

(b) $\mathtt{T}^{\text{out}}$ by GPT4TaskSyn

```
def Run(){
  RepeatUntil(goal){
    If(pathLeft){
      turnLeft
      move
    }
    Else{
      move
    }
  }
}
```

(c) $\mathtt{C}^{\text{out}}$ by GPT4TaskSyn

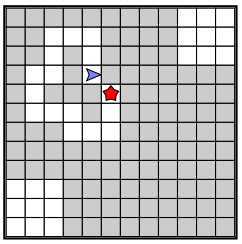


Allowed blocks: {move, turnLeft, turnRight, RepeatUntil, IfElse}
Maximum size: 7

(d) $\mathtt{T}^{\text{out}}$ by BaseTaskSyn

```
def Run(){
  turnLeft
  turnLeft
  RepeatUntil(goal){
    If(pathRight){
      move
    }
    Else{
      move
    }
  }
}
```

(e) $\mathtt{C}^{\text{out}}$ by BaseTaskSyn

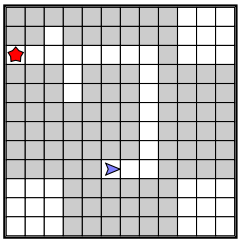


Allowed blocks: {move, turnLeft, turnRight, RepeatUntil, IfElse}
Maximum size: 7

(f) $\mathtt{T}^{\text{out}}$ by NeurTaskSyn

```
def Run(){
  RepeatUntil(goal){
    If(pathRight){
      turnRight
    }
    Else{
      turnLeft
      move
    }
    move
  }
}
```

(g) $\mathtt{C}^{\text{out}}$ by NeurTaskSyn



Allowed blocks: {move, turnLeft, turnRight, RepeatUntil, IfElse}
Maximum size: 5

(h) $\mathtt{T}^{\text{out}}$ by Expert

```
def Run(){
  RepeatUntil(goal){
    If(pathAhead){
      move
    }
    Else{
      turnLeft
    }
  }
}
```

(i) $\mathtt{C}^{\text{out}}$ by Expert

Figure 11: Illustration containing the tuple $\mathtt{T}^{\text{out}}$ for each technique, along with $\mathtt{C}^{\text{out}}$, for this HoCMaze example.

C.2 Example for Karel in Figure 9

We present $\texttt{T}^{\text{out}}$, along with $\texttt{C}^{\text{out}}$ for each of the techniques with ψ^{in} as input in Figure 13; this figure expands on Figure 9 with additional details. We give additional explanations regarding how each of the techniques' output respects or not metrics O1-O3 (see Sections 2 and 5) in Figure 12.

Generation/adjustment for GPT4TaskSyn in Figure 9. For this example, we set $\psi^{\text{in}}_{\text{IO}}$ as an empty 12x12 grid and query GPT-4. We discuss the results obtained by GPT4TaskSyn-*converse*.

Generation/adjustment for BaseTaskSyn and NeurTaskSyn in Figure 9. The neural model for puzzle generation is trained on 16x16 grids, yet the symbolic engine can support the existence of pre-initialized grids. Thus, we mask the upper-left part of the grid (4 rows and 4 columns), obtaining the 12x12 workspace for the technique.

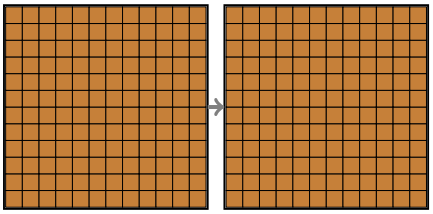
Generation/adjustment for Expert in Figure 9. The output of Expert for this example is based on the Karel:Stairway task.

Technique	**O1:Validity**	**O2:Concepts**	**O3:Trace**	**O4:Overall**	$\texttt{C}^{\text{out}}$ **solves** $\texttt{T}^{\text{out}}$
GPT4TaskSyn	0	0	0	0	0
BaseTaskSyn	1	0	1	0	1
NeurTaskSyn	1	1	1	1	1
Expert	1	1	1	1	1

Figure 12: Scores showing whether the output $\texttt{T}^{\text{out}}$ for ψ^{in} of each technique respects the six metrics O1-O4, with the additional $\texttt{C}^{\text{out}}$ solves $\texttt{T}^{\text{out}}$ metric, for this Karel example.

We provide explanations for each 0 entry for the objectives O1-O3 in Figure 12:

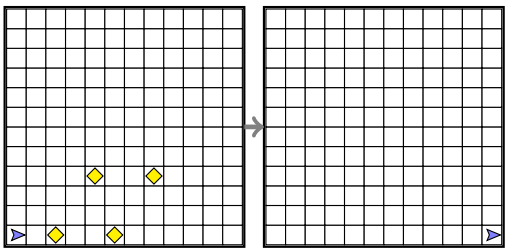
- GPT4TaskSyn for Objective O1: $\texttt{T}^{\text{out}}_{\text{IO}}$ cannot be solved with any code respecting $\texttt{T}^{\text{out}}_{\text{code}}$, hence O1 is 0.
- GPT4TaskSyn for Objective O2: $\texttt{T}^{\text{out}}_{\text{IO}}$ cannot be solved with any code respecting $\texttt{T}^{\text{out}}_{\text{code}}$, hence O2 is 0.
- GPT4TaskSyn for Objective O3: $\texttt{T}^{\text{out}}_{\text{IO}}$ cannot be solved with any code respecting $\texttt{T}^{\text{out}}_{\text{code}}$, hence O3 is 0.
- GPT4TaskSyn for $\texttt{C}^{\text{out}}$ solves $\texttt{T}^{\text{out}}$: The generated code $\texttt{C}^{\text{out}}$ does not solve $\texttt{T}^{\text{out}}$, i.e., the pregrid is not transformed into the postgrid after code execution.
- BaseTaskSyn for Objective O2: The employed `If` block is not required. This implies that there is a solution code `C` which has a lower complexity, i.e., a smaller depth and less constructs than required by $\psi^{\text{in}}_{\text{sketch}}$, hence O2 is 0.



```
def RUN(){
  a blocks
  WHILE(b){
    a blocks
    IF(b){
      a blocks
    }
    a blocks
  }
  a blocks
}
```

a blocks is a body of basic action blocks from the set {move, turnLeft, turnRight, pickMarker, putMarker}
b is a boolean condition from {pathAhead, pathLeft, pathRight, no-pathAhead, markerPresent, no-markerPresent}
Number of blocks should be ≤ 10

(a) Task synthesis specification ψ^{in}

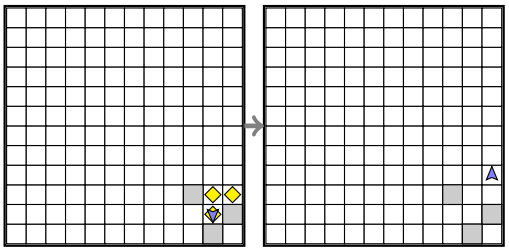


Allowed blocks: {move, turnLeft, turnRight, pickMarker, putMarker, WHILE, IF}
Maximum size: 10

(b) $\mathtt{T}^{\text{out}}$ by GPT4TaskSyn

```
def RUN(){
  WHILE(pathAhead){
    IF(markerPresent){
      pickMarker
    }
    move
  }
}
```

(c) $\mathtt{C}^{\text{out}}$ by GPT4TaskSyn

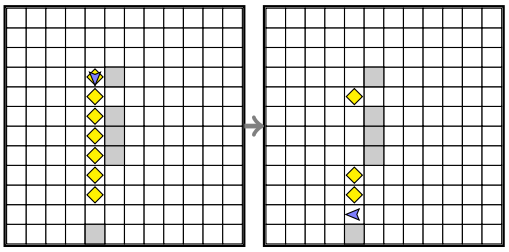


Allowed blocks: {move, turnLeft, turnRight, pickMarker, putMarker, WHILE, IF}
Maximum size: 10

(d) $\mathtt{T}^{\text{out}}$ by BaseTaskSyn

```
def RUN(){
  WHILE(no-pathAhead){
    IF(pathRight){
      pickMarker
    }
    turnRight
    turnRight
    move
    turnLeft
  }
  pickMarker
  move
}
```

(e) $\mathtt{C}^{\text{out}}$ by BaseTaskSyn

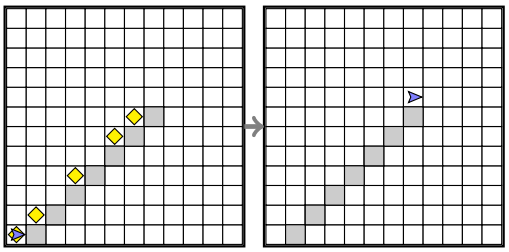


Allowed blocks: {move, turnLeft, turnRight, pickMarker, putMarker, WHILE, IF}
Maximum size: 7

(f) $\mathtt{T}^{\text{out}}$ by NeurTaskSyn

```
def RUN(){
  WHILE(pathAhead){
    pickMarker
    IF(pathLeft){
      putMarker
    }
    move
  }
  turnRight
}
```

(g) $\mathtt{C}^{\text{out}}$ by NeurTaskSyn



Allowed blocks: {move, turnLeft, turnRight, pickMarker, putMarker, WHILE, IF}
Maximum size: 8

(h) $\mathtt{T}^{\text{out}}$ by Expert

```
def RUN(){
  WHILE(no-pathAhead){
    IF(markerPresent){
      pickMarker
    }
    turnLeft
    move
    turnRight
    move
  }
}
```

(i) $\mathtt{C}^{\text{out}}$ by Expert

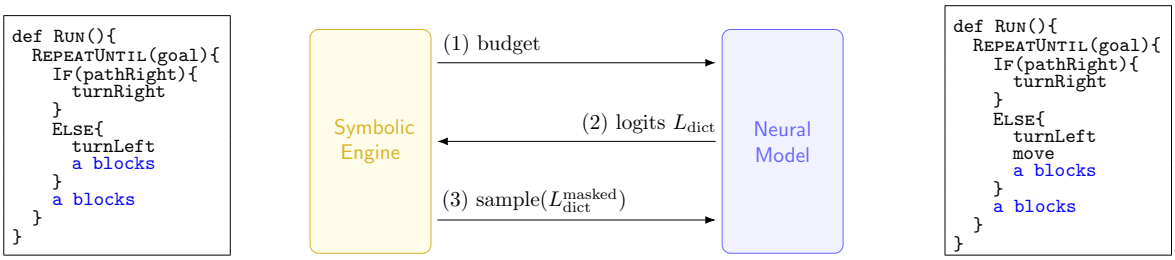
Figure 13: Illustration containing the tuple $\mathtt{T}^{\text{out}}$ for each technique, along with $\mathtt{C}^{\text{out}}$, for this Karel example.

D Our Synthesis Technique NeurTaskSyn: Details

Next, we give additional details regarding each module of our architecture. We present the interaction between the neural models and the underlying symbolic engines, the neural architecture, and the training procedures.

D.1 Generating the Solution Code $\mathtt{C}^{\text{out}}$

Code generator visualization. We describe the interaction between the neural model and the underlying symbolic engine for the code generator. For better understanding, we use one concrete example, illustrated in Figure 14. We consider the AST at time t as presented in Figure 14a, where the previously taken decision was the addition of the `turnLeft` token. The symbolic engine continues its depth-first traversal of the AST and now needs to take the next decision for the subsequent '`a blocks`'. This is achieved by interrogating the neural component; interaction demonstrated in Figure 14b. We introduce the notion of a *budget*, which represents the number of available blocks that can be added to the AST so that $\psi^{\text{in}}_{\text{size}}$ is respected; in our example, the remaining budget is 2. It is passed as input for the neural model at time t. The neural model, based on the budget and its internal state, which keeps track of the previously taken decisions, outputs a logit for each decision, i.e., a set of logits L_{dict}. The symbolic engine accepts L_{dict} and masks them according to the rules in the DSL, thus obtaining $L^{\text{masked}}_{\text{dict}}$. In our example, the only values in $L^{\text{masked}}_{\text{dict}}$ that are not masked are those of basic action blocks (i.e., `move`, `turnLeft`, `turnRight`) and the token that represents the end of the `Else` body. After mapping the logits to a probability distribution, the symbolic engine proceeds to sample a decision from it. In our example, `move` is sampled. The decision is passed to the neural model to update its internal state. The symbolic engine then updates the AST with the taken decision (i.e., `move`), thus obtaining the updated version of the AST for the next step at time $t+1$, illustrated in Figure 14c. We generalize this process to every decision that needs to be taken while traversing the AST.



(a) Code status at time t (b) Interaction at time t (c) Code status at time $t+1$

Figure 14: Visualization of the interaction process between the neural model and the symbolic engine in the code generator component of NeurTaskSyn. (a) shows the AST at time t, where the first '`a blocks`' needs to be decided. (b) shows the interaction between the symbolic engine and the neural model at time t, where the symbolic engine first passes the *budget* (available blocks) to the neural model, the neural model computes the logits for all the tokens in the dictionary L_{dict} and passes them back to the symbolic engine, which finally masks them obtaining $L^{\text{masked}}_{\text{dict}}$, applies softmax to obtain a probability and samples the next action, sending it to the neural model. (c) shows the AST at time $t+1$, where the sampled `move` was integrated.

Imitation learning procedure. We will now give details about the learning procedure we used for the neural model. Given the fact that dataset $\mathbb{D} := \{\psi^{\text{in}}\}$ is accompanied by example codes, i.e., $\mathtt{C}^{\text{in}}$ for each ψ^{in}, we employ an imitation (supervised) learning approach, similar to (Devlin et al., 2017; Bunel et al., 2018). Thus, for each decision, we compute the cross-entropy with respect to the target decision. We force the agent to take the target decision afterward so the generated code does not digress from our example code.

Neural architecture. Here, we present in detail the architecture of the neural model we employ for code generation. Similar to (Bunel et al., 2018), we employ an LSTM-based (Hochreiter & Schmidhuber, 1997) recurrent neural network. We first convert code tokens to indexes based on a dictionary, then we pass them through an embedding layer. We do the same with the numeric representation of the *budget* (introduced previously). We concatenate both embeddings and pass them through a two-layer LSTM. Last, we convert the output of the LSTM to logits for each entry in the dictionary using a linear layer. The architecture can be observed in Figure 15.

Input	Code token categorical (0-58)	Budget ordinal (0-16)
Embedding	Size = 256	Size = 16
LSTM 1	Hidden dim = 256	
LSTM 2	Hidden dim = 256	
Linear	Hidden dim $\times$ Dict size = 256 $\times$ 59	

Figure 15: Architecture of the neural model used by the code generator.

D.2 Generating the Visual Puzzle $\mathsf{T}_{\mathsf{IO}}^{\mathsf{out}}$

Reinforcement learning procedure. We describe further details necessary for training our reinforcement learning (RL) agent. To learn a policy, we use policy gradient methods. These methods generally learn by using gradient ascent, thus updating the parameters θ of the parameterized policy $\pi_\theta(a|s)$ to increase the expected reward of the policy in the MDP. Naturally, a neural network can be used to learn the policy, where θ represents the network's weights. The network would take action a and state s as input, outputting a logit $H_\theta(a|s)$. Given the logits, we map them to a probabilistic distribution using softmax: $\pi_\theta(a|s) = \frac{\exp(H_\theta(a|s))}{\sum_{a' \in A_s} \exp(H_\theta(a'|s))}$. We use an actor-critic policy gradient method for agent training (Sutton & Barto, 2018). We denote with $\hat{v}(s, w)$ the value for state s predicted by the critic with parameters w. As we operate on batches, the parameters of both the actor and the critic remain unchanged until a buffer is filled with a fixed number of episodes. Thus, for an initial state s_0 (i.e., an empty or pre-initialized task and a code, with the emulator reset), we execute the existing policy π_θ until the buffer is filled, generating several sequences of experience as tuples $(s_t, a_t, r_t)_{t=0..T}$, where T represents a variable episode length. Thus, the losses for an episode are computed as a sum over the timesteps $t \in [0, T]$ as follows, for the actor (Equation 1) and for the critic (Equation 2, employing the smooth L1 loss, denoted as $\text{L1}_{\text{smooth}}$):

$$\text{Loss}_\theta = \sum_{t=0}^{T} \left(\sum_{\tau=t}^{T} r_\tau - \hat{v}(s_t, w) \right) \cdot \nabla_\theta \log(\pi_\theta(a_t|s_t)) \tag{1}$$

$$\text{Loss}_w = \sum_{t=0}^{T} \text{L1}_{\text{smooth}} \left(\sum_{\tau=t}^{T} r_\tau,\ \hat{v}(s_t, w) \right) \tag{2}$$

Finally, θ and w are updated by using the computed losses over the entire batch, multiplied with a learning rate.

Neural architecture. We describe the architecture of the CNN-based neural model used by the puzzle generator. We employ a similar architecture for both the HoCMaze and Karel domains, as presented in Figure 16. Only the input size for the grid (i.e., D $\times$ 16 $\times$ 16, where D = 12 for HoCMaze and D = 14 for Karel) and code features (i.e., F, where F = 9 for HoCMaze and F = 12 for Karel) differ. We process the grid by 3 CNN blocks (i.e., one block composed of Conv2D, ReLU, and MaxPooling2D layers), after which we apply 5 fully connected (linear) layers, thus obtaining the grid embedding. To the grid embedding, we concatenate the code features, which are represented in a binary manner (e.g., increase in coverage, current decision type). We then pass the concatenated tensor through an additional fully-connected layer, and its output is then passed to both the action head and the value head (i.e., necessary for the Actor-Critic algorithm).

Input	D × 16 × 16 Grid	Code features, Size = F
CNN Block 1	Conv2D, kernel size = 3, padding 1, 64 × 64 ReLU MaxPool2D, kernel size = 2, padding 0	
CNN Block 2	Conv2D, kernel size = 3, padding 1, 64 × 64 ReLU MaxPool2D, kernel size = 2, padding 0	
CNN Block 3	Conv2D, kernel size = 3, padding 1, 64 × 64 ReLU MaxPool2D, kernel size = 2, padding 0	
Linear 1	CNN output size (256) × 1024	
Linear 2	1024 × 512	
Linear 3	512 × 256	
Linear 4	256 × 128	
Linear 5	128 × 32	
Linear 6	Concatenated features size (32 + F) × 8	
Linear 7 (Action and Value)	8 × action space	8 × 1

Figure 16: Architecture of the neural model used by the puzzle generator. D denotes the depth of the input grid and F denotes the size of the code features tensor, both different for each of the HoCMaze and Karel domains.

E Experimental Evaluation with Synthetic Task Specifications: Details

In this section, we detail the instantiations of the scoring functions, give more insight into the synthetic dataset creation process, and show the details of the training processes for both the code generator and the task generator.

E.1 Scoring Function $\mathcal{F}_{\text{score}}$

Next, we describe the two instantiations for $\mathcal{F}_{\text{score}}$ as used in the two domains HoCMaze and Karel. We adopt a scoring function similar to that of (Ahmed et al., 2020), where $\mathcal{F}_{\text{score}}$ is used for guiding a Monte Carlo Tree Search, as an evaluation function that describes the desired properties of their system's output. We note that our method can work with any other instantiation of the scoring function $\mathcal{F}_{\text{score}}$. The instantiations we use for $\mathcal{F}_{\text{score}}$ for each of the HoCMaze and Karel domains are defined in Equations 3 and 4 and are comprised of different components: (i) $\mathcal{F}_{\text{cov}}(\mathtt{T}^{\text{out}}_{\text{IO}}, \mathtt{C}^{\text{out}}) \in [0,1]$ computes the coverage ratio, i.e., ratio of executed blocks to total number of blocks; (ii) $\mathcal{F}_{\text{sol}}(\mathtt{T}^{\text{out}}_{\text{IO}}, \mathtt{C}^{\text{out}}) \in \{0,1\}$ evaluates to 1 if $\mathtt{C}^{\text{out}}$ correctly solves $\mathtt{T}^{\text{out}}_{\text{IO}}$, i.e., no crashing, reaching the goal/converting the pre-grid to the post-grid; (iii) $\mathcal{F}_{\text{nocross}}(\mathtt{T}^{\text{out}}_{\text{IO}}, \mathtt{C}^{\text{out}}) \in [0,1]$ computes the ratio of cells visited exactly once with regard to the total number of visited cells; (iv) $\mathcal{F}_{\text{nocut}}(\mathtt{T}^{\text{out}}_{\text{IO}}, \mathtt{C}^{\text{out}}) \in \{0,1\}$ evaluates to 0 if there is a shortcut sequence comprised of basic actions; (v) $\mathcal{F}_{\text{notred}}(\mathtt{T}^{\text{out}}_{\text{IO}}, \mathtt{C}^{\text{out}}) \in \{0,1\}$ evaluates to 0 if there are redundant action sequences in $\mathtt{C}^{\text{out}}$, e.g., sequences like `turnLeft`, `turnRight`, or if the codes obtained by eliminating one action, loop or conditional from $\mathtt{C}^{\text{out}}$ solves $\mathtt{T}^{\text{out}}_{\text{IO}}$; (vi) $\mathcal{F}_{\text{qual}}(\mathtt{T}^{\text{out}}_{\text{IO}}, \mathtt{C}^{\text{out}}) \in [0,1]$ evaluates the visual quality of $\mathtt{T}^{\text{out}}_{\text{IO}}$ as per Equation 5; (vii) $\mathcal{F}_{\text{cutqual}}(\mathtt{T}^{\text{out}}_{\text{IO}}, \mathtt{C}^{\text{out}}) \in [0,1]$ evaluates visual quality of the shortest path made only of basic actions, similar to $\mathcal{F}_{\text{qual}}$. We set $\alpha_1 = \alpha_2 = \frac{1}{2}$ and $\alpha_3 = \alpha_4 = \alpha_5 = \frac{1}{3}$.

$$\begin{aligned}\mathcal{F}^{\text{HoCMaze}}_{\text{score}}(\mathtt{T}^{\text{out}}, \mathtt{C}^{\text{out}}) = &\mathbf{1}\Big[\mathcal{F}_{\text{cov}}(\mathtt{T}^{\text{out}}_{\text{IO}}, \mathtt{C}^{\text{out}}) = 1, \mathcal{F}_{\text{sol}}(\mathtt{T}^{\text{out}}_{\text{IO}}, \mathtt{C}^{\text{out}}) = 1, \mathcal{F}_{\text{nocross}}(\mathtt{T}^{\text{out}}_{\text{IO}}, \mathtt{C}^{\text{out}}) = 1, \\ &\mathcal{F}_{\text{nocut}}(\mathtt{T}^{\text{out}}_{\text{IO}}, \mathtt{C}^{\text{out}}) = 1, \mathcal{F}_{\text{notred}}(\mathtt{T}^{\text{out}}_{\text{IO}}, \mathtt{C}^{\text{out}}) = 1\Big] \cdot \\ &\Big[\alpha_1 \mathcal{F}_{\text{cov}}(\mathtt{T}^{\text{out}}_{\text{IO}}, \mathtt{C}^{\text{out}}) + \alpha_2 \mathcal{F}_{\text{qual}}(\mathtt{T}^{\text{out}}_{\text{IO}}, \mathtt{C}^{\text{out}})\Big]\end{aligned} \tag{3}$$

$$\begin{aligned}\mathcal{F}^{\text{Karel}}_{\text{score}}(\mathtt{T}^{\text{out}}, \mathtt{C}^{\text{out}}) = &\mathbf{1}\Big[\mathcal{F}_{\text{cov}}(\mathtt{T}^{\text{out}}_{\text{IO}}, \mathtt{C}^{\text{out}}) = 1, \mathcal{F}_{\text{sol}}(\mathtt{T}^{\text{out}}_{\text{IO}}, \mathtt{C}^{\text{out}}) = 1, \mathcal{F}_{\text{nocross}}(\mathtt{T}^{\text{out}}_{\text{IO}}, \mathtt{C}^{\text{out}}) = 1, \\ &\mathcal{F}_{\text{nocut}}(\mathtt{T}^{\text{out}}_{\text{IO}}, \mathtt{C}^{\text{out}}) = 1, \mathcal{F}_{\text{notred}}(\mathtt{T}^{\text{out}}_{\text{IO}}, \mathtt{C}^{\text{out}}) = 1\Big] \cdot \\ &\Big[\alpha_3 \mathcal{F}_{\text{cov}}(\mathtt{T}^{\text{out}}_{\text{IO}}, \mathtt{C}^{\text{out}}) + \alpha_4 \mathcal{F}_{\text{qual}}(\mathtt{T}^{\text{out}}_{\text{IO}}, \mathtt{C}^{\text{out}}) + \alpha_5 \mathcal{F}_{\text{cutqual}}(\mathtt{T}^{\text{out}}_{\text{IO}}, \mathtt{C}^{\text{out}})\Big]\end{aligned} \tag{4}$$

We use the same measure of visual quality for both domains, keeping into account the number of moves, turns, segments, long-segments, and turn-segments, as explained next. More specifically, segments and long-segments correspond to consecutive sequences of 'moves' containing more than 3 and 5 actions, respectively; turn-segments correspond to consecutive sequences of 'turnLeft' or 'turnRight' containing more than 3 actions. The formula for $\mathcal{F}_{\text{qual}}$ is given in Equation 5 below; we also clip the values for each counter # w.r.t. its corresponding normalization factor (not depicted here for brevity).

$$\begin{aligned}\mathcal{F}_{\text{qual}}(\mathtt{T}^{\text{out}}, \mathtt{C}^{\text{out}}) = &\frac{3}{4} \cdot \left(\frac{1}{4} \cdot \left(\frac{\#\text{moves}}{2n} + \frac{\#\text{turns}}{n} + \frac{\#\text{segments}}{n/2} + \frac{\#\text{long-segments}}{n/3}\right)\right) + \\ &\frac{1}{4} \cdot \left(1 - \frac{\#\text{turn-segments}}{n/2}\right)\end{aligned} \tag{5}$$

E.2 Synthetic Task Specifications

Domain-specific elements. As introduced in Sections 2 and 4, we use two DSLs shown in Figures 17a and 17b, adapted from the DSLs in (Bunel et al., 2018; Ahmed et al., 2020).

```
code C      := def Run () do y
rule y      := s | g | s; g
rule s      := a | s; s | If (b) do s
             | If (b) do s Else s
             | Repeat (x) do s
rule g      := RepeatUntil (goal) do s
action a    := move | turnLeft | turnRight
bool b      := pathAhead | pathLeft | pathRight
iter x      := 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10
```

(a) DSL for HoCMaze domain

```
code C      := def Run () do s
rule s      := a | s; s | If (b) do s | If (b) do s Else s
             | While (b) do s | Repeat (x) do s
action a    := move | turnLeft | turnRight
             | putMarker | pickMarker
bool b      := pathAhead | pathLeft | pathRight
             | no-pathAhead | markerPresent | no-markerPresent
iter x      := 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10
```

(b) DSL for Karel domain

Domain	**All**	**Easy** (depth, constructs)			**Hard** (depth, constructs)	
		$(1,0)$	$(2,1)$	$(2,2)$	$(3,2)$	$(3,3)$
HoCMaze	$1,016$	183	69	47	136	581
Karel	$1,027$	300	155	277	295	0

(c) Dataset of synthetic task specifications

Figure 17: **(a)** Synthetic datasets used for training/evaluation in Section 4). **(b)** DSLs for two domains.

Algorithm 1: Specification Dataset Collection Procedure

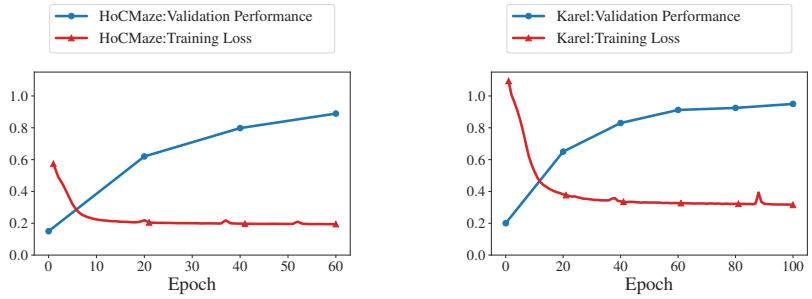
Input: list $\mathbb{S}$ of tuples (code structure s, required size l); maximum candidate set size m;
$\mathbb{D} \leftarrow \emptyset$; `/* Dataset initialized to empty set */`
foreach $(s, l) \in \mathbb{S}$ **do**
  $\mathbb{C} \leftarrow \emptyset$; `/* Candidate set initialized to empty set */`
  while size($\mathbb{C}$) $< m$ **do**
    code $\leftarrow$ GenerateCode(s);
    task $\leftarrow$ TaskOracle(code);
    score $\leftarrow \mathcal{F}_{\text{score}}$(task, code);
    if score > 0 **then**
      add (code, task, score) to $\mathbb{C}$;
  sort $\mathbb{C}$ according to the score, in decreasing order;
  counter $\leftarrow 0$;
  while counter $< l$ and $\mathbb{C} \neq \emptyset$ **do**
    (code, task, score) $\leftarrow$ Pop($\mathbb{C}$);
    accept $\leftarrow$ Inspect(task, code) ; `/* Inspection step */`
    if accept **then**
      $\psi \leftarrow$ ExtractSpecs(code);
      add ψ to $\mathbb{D}$;
      counter $\leftarrow$ counter+1;

Output: Dataset $\mathbb{D}$;

Dataset. We follow Algorithm 1 to create dataset $\mathbb{D}$. For each code structure, we generate a set of candidate codes and obtain an oracle task for these codes. We filter them out if a low-quality task is obtained, supplementing this filtering with an additional inspection step. This inspection step is necessary because semantic irregularities (e.g., `IfElse` with the same `If` and `Else` bodies) can get past the previous filtering step. As the compute and implementation efforts are larger for an automatic system that would detect such irregularities, which are easy to spot, we opt for a direct inspection step. Figure 17c provides a summary of datasets $\mathbb{D}$ for each domain.

E.3 Training Process

Training the code generator. We employ a standard approach, using an imitation (supervised) form of learning, with a cross-entropy loss for an LSTM-based architecture (see Appendix D) (Devlin et al., 2017; Bunel et al., 2018). We augment $\mathbb{D}^{\text{train}}$ by adding all the possible combinations of construct instantiations for a given code. The training plots and the hyperparameters used can be seen in Figure 18. We report the validation performance smoothed via an exponential decay function, and the batch loss averaged over one epoch.



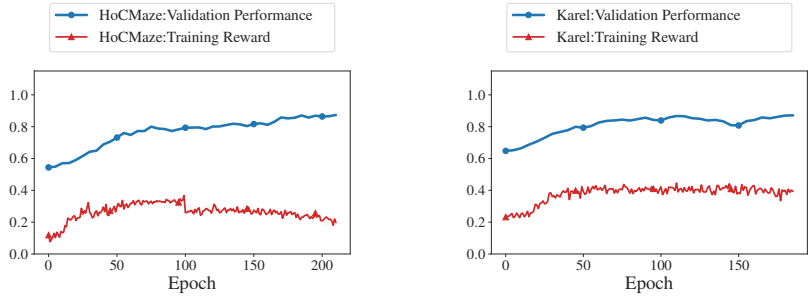
(a) HoCMaze (b) Karel

HyperParameter	HoCMaze	Karel
Max. Epochs	60	100
Batch Size	32	32
Learning Rate	5×10^{-4}	5×10^{-4}
Dict. Size	59	59
Max. Blocks	17	17

(c) Hyperparameters

Figure 18: Illustration of training details for the code generator. (a) and (b) show the training curves with mean epoch loss and validation performance, based on metric $\mathcal{M}$, for both the HoCMaze and Karel domains. (c) shows the hyperparameters employed for the code generator training.

Training the puzzle generator. We use an RL procedure, using the instantiations of $\mathcal{F}_{\text{score}}$ as rewards. We augment the RL training set with additional codes produced by the previously trained code generator. To encourage higher quality tasks, we use a form of curriculum as follows: after a certain epoch, we give a reward larger than 0 only if the ratio between the scores of the output task and the TaskOracle's task is larger than a factor $\widehat{\lambda}_2$; we gradually increase $\widehat{\lambda}_2$ from 0.8 to 0.9. For Karel, we also employ a temperature parameter during training, encouraging exploration during inference. The training plots and the hyperparameters used can be seen in Figures 19. We report the validation performance (i.e., same metric employed for NeurPuzzleGen) smoothed via an exponential decay function, and the batch reward averaged over one epoch.



(a) HoCMaze (b) Karel

HyperParameter	HoCMaze	Karel
Max. Epochs	500	500
Batch Size	32	32
Learning Rate	10^{-4}	10^{-4}
Temperature	1	0.5
Curriculum	True	True

(c) Hyperparameters

Figure 19: Illustration of training details for the puzzle generator. (a) and (b) show the training curves with mean epoch reward and validation performance, based on metric $\mathcal{M}$ for both the HoCMaze and Karel domains. A form of curriculum learning was employed, which explains the lack of general monotonicity for the reward. (c) shows the hyperparameters employed for the puzzle generator training.

Further implementation details. We limit the number of possible initial locations for a grid to one representative per quadrant. In total, we consider 5 quadrants (i.e., top-left, bottom-left, center, top-right, bottom-right). We do this to limit the action space to a more tractable amount for a variable grid size. With 5 quadrants and 4 possible orientations, this leads to $5 \times 4 = 20$ possible initial location/orientation pairs, offering already enough variability.

Detailed results on synthetic task specifications. Figure 20 reports evaluation results for different techniques for a fixed number of code/puzzle rollouts across two domains and segments, complementary to Figure 5.

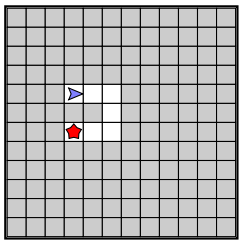
Technique	**HoCMaze**			**Karel**		
	All	Easy	Hard	All	Easy	Hard
BaseTaskSyn$_{c:5,p:10}$	13.6 (1.3)	46.2 (4.1)	2.0 (1.3)	35.7 (1.0)	49.3 (0.9)	10.9 (2.7)
NeurTaskSyn$_{c:5,p:10}$	81.4 (3.7)	100.0 (0.0)	74.3 (5.1)	92.6 (1.4)	100.0 (0.0)	79.3 (3.9)
BasePuzzleGen$_{c:FIX,p:10}$	55.6 (1.8)	91.7 (2.4)	41.9 (2.3)	71.8 (3.8)	86.6 (3.8)	45.0 (3.9)
NeurPuzzleGen$_{c:FIX,p:10}$	78.4 (2.5)	100.0 (0.0)	70.3 (3.4)	79.8 (0.6)	92.0 (1.3)	57.7 (1.8)

Figure 20: Results on synthetic task specifications for HoCMaze and Karel; see Figure 17c and Section 4.

F Experimental Evaluation with Real-World Task Specifications: Details

F.1 Real-World Task Specifications

In Figures 21 and 22 below, we list source tasks T and codes C for 10 task specifications mentioned in Figure 6.



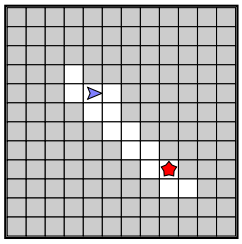
Allowed blocks: {move, turnLeft, turnRight, Repeat}

Maximum size: 5

(a) Source T for ψ_0

```
def Run(){
  Repeat(3){
    move
    move
    turnRight
  }
}
```

(b) Source C for ψ_0



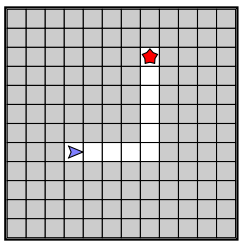
Allowed blocks: {move, turnLeft, turnRight, RepeatUntil}

Maximum size: 6

(c) Source T for ψ_1

```
def Run(){
  RepeatUntil(goal){
    turnRight
    move
    turnLeft
    move
  }
}
```

(d) Source C for ψ_1



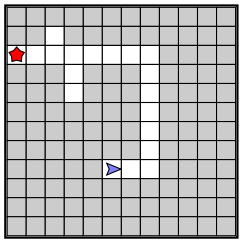
Allowed blocks: {move, turnLeft, turnRight, Repeat}

Maximum size: 6

(e) Source T for ψ_2

```
def Run(){
  Repeat(4){
    move
  }
  turnLeft
  Repeat(5){
    move
  }
}
```

(f) Source C for ψ_2



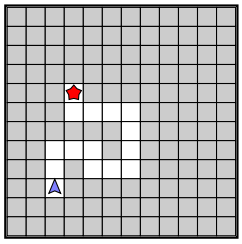
Allowed blocks: {move, turnLeft, turnRight, RepeatUntil, IfElse}

Maximum size: 5

(g) Source T for ψ_3

```
def Run(){
  RepeatUntil(goal){
    If(pathAhead){
      move
    }
    Else{
      turnLeft
    }
  }
}
```

(h) Source C for ψ_3



Allowed blocks: {move, turnLeft, turnRight, RepeatUntil, If}

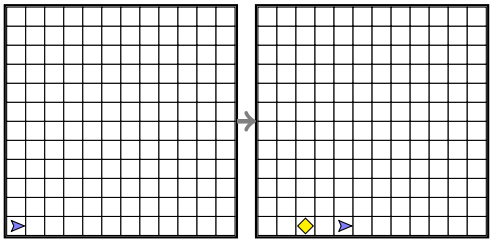
Maximum size: 7

(i) Source T for ψ_4

```
def Run(){
  RepeatUntil(goal){
    move
    If(pathLeft){
      turnLeft
    }
    If(pathRight){
      turnRight
    }
  }
}
```

(j) Source C for ψ_4

Figure 21: Overview of HoCMaze sources.



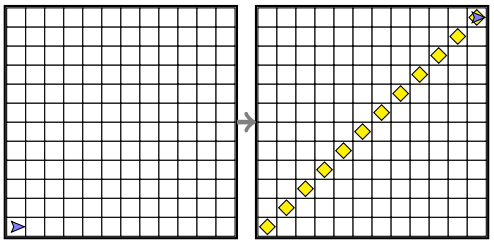
Allowed blocks: {move, turnLeft, turnRight, pickMarker, putMarker}

Maximum size: 6

```
def Run(){
  move
  move
  putMarker
  move
  move
}
```

(a) Source T for ψ_5

(b) Source C for ψ_5



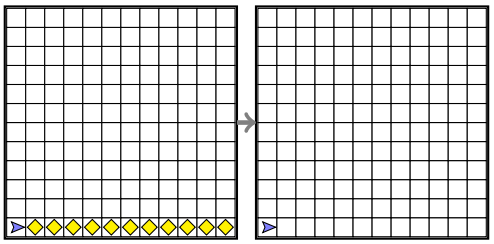
Allowed blocks: {move, turnLeft, turnRight, pickMarker, putMarker, While}

Maximum size: 8

```
def Run(){
  putMarker
  While(pathAhead){
    move
    turnLeft
    move
    turnRight
    putMarker
  }
}
```

(c) Source T for ψ_6

(d) Source C for ψ_6



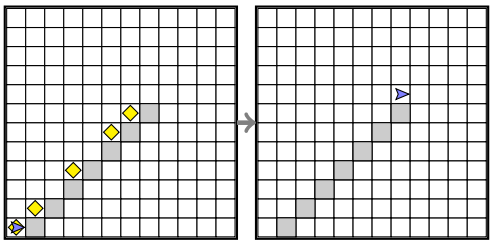
Allowed blocks: {move, turnLeft, turnRight, pickMarker, putMarker, While}

Maximum size: 10

```
def Run(){
  While(pathAhead){
    move
    pickMarker
  }
  turnLeft
  turnLeft
  While(pathAhead){
    move
  }
  turnLeft
  turnLeft
}
```

(e) Source T for ψ_7

(f) Source C for ψ_7



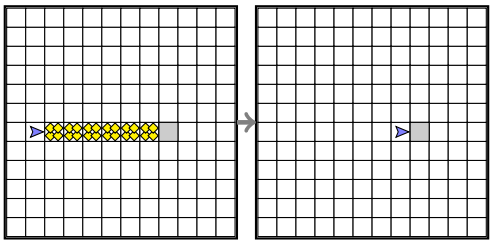
Allowed blocks: {move, turnLeft, turnRight, pickMarker, putMarker, While, If}

Maximum size: 8

```
def Run(){
  While(no-pathAhead){
    If(markerPresent){
      pickMarker
    }
    turnLeft
    move
    turnRight
    move
  }
}
```

(g) Source T for ψ_8

(h) Source C by ψ_8



Allowed blocks: {move, turnLeft, turnRight, pickMarker, putMarker, While, Repeat}

Maximum size: 5

```
def Run(){
  While(pathAhead){
    move
    Repeat(4){
      pickMarker
    }
  }
}
```

(i) Source T for ψ_9

(j) Source C by ψ_9

Figure 22: Overview of Karel sources.

F.2 GPT4TaskSyn

We describe next the details of our interaction with GPT-4 for generating the visual puzzles $\mathtt{T}_{\text{IO}}^{\text{out}}$ and the intermediate code $\mathtt{C}^{\text{out}}$ via the GPT4TaskSyn techniques. Our interaction is conducted through the platform (OpenAI, 2023a). We try several strategies and prompts to make GPT-4 work for synthesizing visual programming tasks, as it tends to struggle with logical and spatial reasoning. Thus, we opt for a two-stage task synthesis process which works the best. We first ask GPT-4 to generate a code $\mathtt{C}^{\text{out}}$ for ψ^{in}, by using 5 separate queries.

For the GPT4TaskSyn-*converse* technique, we start with an initial prompt and then use follow-up prompts to fix any mistakes, as GPT-4 occasionally ignores part of the specifications. The initial and follow-up prompts used for generating $\mathtt{C}^{\text{out}}$ are presented in Figures 23a and 24a. We select the best code generated during the 5 separate queries based on our expertise. The second stage comprises of additional 5 separate queries for generating $\mathtt{T}_{\text{IO}}^{\text{out}}$ for the selected code $\mathtt{C}^{\text{out}}$. Again, we start with an initial prompt and then use follow-up prompts to fix any issues. The follow-up prompts are necessary because GPT-4 tends to struggle with spatial orientation and with the relationship between $\mathtt{C}^{\text{out}}$ and $\mathtt{T}_{\text{IO}}^{\text{out}}$. The initial and follow-up prompts used for generating $\mathtt{T}_{\text{IO}}^{\text{out}}$ are presented in Figures 23b and 24b. Similar to the code selection process, we select the best visual puzzle generated during the 5 separate queries based on our expertise. Once we get $\mathtt{C}^{\text{out}}$ and $\mathtt{T}_{\text{IO}}^{\text{out}}$, we set $\mathtt{T}_{\text{code}}^{\text{out}}$ as for BaseTaskSyn and NeurTaskSyn.

For the GPT4TaskSyn-*fewshot*, technique, we employ a few-shot approach for both stages, by first giving 3 synthesis examples (i.e., for code and task synthesis, respectively). We do not use follow-up prompts here. The few-shot prompts are presented in Figures 25 and 26.

Code: Initial prompt

I am working in the block-based visual programming domain of Hour of Code: Maze Challenge from code.org. In this domain, the following types of coding blocks are available:

- Basic action blocks: move forward, turn left, turn right.

- Boolean conditions: path ahead, path left, path right.

- Loops: repeatUntil(goal), repeat(int).

- Conditionals: if(boolean), if(boolean)else.

In this domain, a task is represented as an 8x8 visual grid that contains WALL cells, FREE cells, AVATAR (with specific location and direction), and GOAL. We represent a task's 8x8 visual grid with the following symbols.

represents a WALL cell.

+ represents a FREE cell.

* represents GOAL.

E represents AVATAR's location facing East direction.

W represents AVATAR's location facing West direction.

N represents AVATAR's location facing North direction.

S represents AVATAR's location facing South direction.

Below, I am giving you a program structure. Can you generate a code that respects this program structure?

— Structure —

[SKETCH]

You should not change the structure. This means that you shouldn't add or remove any loops (e.g., repeatUntil(goal), repeat(int)) and conditionals (e.g., if(boolean), if(boolean)else). The program needs to be valid, meaning that bodies of constructs cannot remain empty. To complete this given structure, you can use basic action blocks, boolean conditions, and iteration numbers that are available in the Hour of Code: Maze Challenge programming.

— Code —

Code: Follow-up prompt in case of constructs changed

Your code does not follow the program structure I have given. You shouldn't add or remove any loops (e.g., repeatUntil(goal), repeat(int)) and conditionals (e.g., if(boolean), if(boolean)else). Can you try to generate a new code for the same structure?

Code: Follow-up prompt for any other issues

Your code could be improved! You can think of producing a better code by reasoning about the AVATAR's actions when the code is executed. Can you try to generate a new code respecting the program structure I have given?

(a) Prompts used for obtaining $\mathtt{C}^{\text{out}}$

Task: Initial prompt

I am working in the block-based visual programming domain of Hour of Code: Maze Challenge from code.org. In this domain, the following types of coding blocks are available:

- Basic action blocks: move forward, turn left, turn right.

- Boolean conditions: path ahead, path left, path right.

- Loops: repeatUntil(goal), repeat(int).

- Conditionals: if(boolean), if(boolean)else.

In this domain, a task is represented as an 8x8 visual grid that contains WALL cells, FREE cells, AVATAR (with specific location and direction), and GOAL. We represent a task's 8x8 visual grid with the following symbols.

represents a WALL cell.

+ represents a FREE cell.

* represents GOAL.

E represents AVATAR's location facing East direction.

W represents AVATAR's location facing West direction.

N represents AVATAR's location facing North direction.

S represents AVATAR's location facing South direction.

Below I am giving you a solution code. Can you generate a task with 8x8 visual grid that would be solved by this code?

— Solution —

[CODE]

The visual grid must contain AVATAR (with specific location and direction) along with GOAL, and can have WALL cells and FREE cells. Number your grid with row numbers (1 to 8) and column numbers (1 to 8). Also, you should tell me the position of AVATAR and GOAL in your generated task so we are sure about the numbering.

You can verify the correctness of your generated task by executing the solution code on your task. A solution code for a task takes AVATAR to GOAL when executed. Note that AVATAR can only move on FREE cells and will crash if it tries to go to a WALL cell. If your generated task is not correct, you should try again to generate a correct task.

— Task —

Task: Follow-up prompt for any issues

Your code does not solve the generated grid. Be careful with the AVATAR as it should reach the goal after the code execution. Keep the code fixed. Can you try to generate a new visual grid and explain your reasoning? Recall that your code, when executed, should take the AVATAR from its initial location to the GOAL.

(b) Prompts used for obtaining the main part of $\mathtt{T}_{\mathrm{IO}}^{\mathrm{out}}$

Figure 23: Prompts used in the implementation of GPT4TaskSyn-*converse* technique for HoCMaze domain.

Code: Initial prompt

I am working in the block-based visual programming domain of Karel programming. In this domain, the following types of coding blocks are available:

- Basic action blocks: move forward, turn left, turn right, pick marker, put marker.

- Boolean conditions: path ahead, path left, path right, marker present, no path ahead, no marker present.

- Loops: while(boolean), repeat(int).

- Conditionals: if(boolean), if(boolean)else.

In this domain, a task is represented as a pair of 10x10 visual pregrid and 10x10 visual postgrid. This pregrid and postgrid contain WALL cells, FREE cells, AVATAR (with specific location and direction), and markers. We represent a task's 10x10 visual pregrid and postgrid with the following symbols.

represents a WALL cell.

+ represents a FREE cell.

m represents a cell with marker.

E represents AVATAR's location on a cell without marker, facing East direction.

W represents AVATAR's location on a cell without marker, facing West direction.

N represents AVATAR's location on a cell without marker, facing North direction.

S represents AVATAR's location on a cell without marker, facing South direction.

Em represents AVATAR's location on a cell with marker, facing East direction.

Wm represents AVATAR's location on a cell with marker, facing West direction.

Nm represents AVATAR's location on a cell with marker, facing North direction.

Sm represents AVATAR's location on a cell with marker, facing South direction.

Below, I am giving you a program structure. Can you generate a code that respects this program structure?

—- Structure —-

[SKETCH]

You should not change the structure. This means that you shouldn't add or remove any loops (e.g., while(boolean), repeat(int)) and conditionals (e.g., if(boolean), if(boolean)else). The program needs to be valid, meaning that bodies of constructs cannot remain empty. To complete this given structure, you can use basic action blocks, boolean conditions, and iteration numbers that are available in Karel programming.

— Code —

Code: Follow-up prompt in case of constructs changed

Your code does not follow the programming structure I have given. You shouldn't add or remove any loops (e.g., while(boolean), repeat(int)) and conditionals (e.g., if(boolean), if(boolean)else). Can you try to generate a new code for the same structure?

Code: Follow-up prompt for any other issues

Your code could be improved! You can think of producing a better code by reasoning about the Karel AVATAR when the code is executed. Can you try to generate a new code?

(a) Prompts used for obtaining $\texttt{C}^{\text{out}}$

Task: Initial prompt

I am working in the block-based visual programming domain of Karel programming. In this domain, the following types of coding blocks are available:

- Basic action blocks: move forward, turn left, turn right, pick marker, put marker.

- Boolean conditions: path ahead, path left, path right, marker present, no path ahead, no marker present.

- Loops: while(boolean), repeat(int).

- Conditionals: if(boolean), if(boolean)else.

In this domain, a task is represented as a pair of 10x10 visual pregrid and 10x10 visual postgrid. This pregrid and postgrid contain WALL cells, FREE cells, AVATAR (with specific location and direction), and markers. We represent a task's 10x10 visual pregrid and postgrid with the following symbols.

represents a WALL cell.

+ represents a FREE cell.

m represents a cell with marker.

E represents AVATAR's location on a cell without marker, facing East direction.

W represents AVATAR's location on a cell without marker, facing West direction.

N represents AVATAR's location on a cell without marker, facing North direction.

S represents AVATAR's location on a cell without marker, facing South direction.

Em represents AVATAR's location on a cell with marker, facing East direction.

Wm represents AVATAR's location on a cell with marker, facing West direction.

Nm represents AVATAR's location on a cell with marker, facing North direction.

Sm represents AVATAR's location on a cell with marker, facing South direction.

Below I am giving you a solution code. Can you generate a task with a pair of 10x10 visual pregrid and 10x10 visual postgrid that would be solved by this code?

— Solution —

[CODE]

Both the visual pregrid and visual postgrid must contain AVATAR (with specific location and direction), and can have WALL cells, FREE cells, and markers. Number your grids with row numbers (1 to 10) and column numbers (1 to 10). Also, you should tell me the position of AVATAR in your generated pregrid and postgrid so we are sure about the numbering.

You can verify the correctness of your generated task by executing the solution code on your task. A solution code for a task transforms the pregrid into the postgrid when executed. Note that AVATAR can only move on FREE cells and will crash if it tries to go to a WALL cell. If your generated task is not correct, you should try again to generate a correct task.

— Task —

Task: Follow-up prompt for any issues

Your code does not solve the generated pregrid and postgrid. Be careful with the AVATAR in the postgrid as it should show the effect of the code execution. Keep the code fixed. Can you try to generate a new visual pregrid and postgrid and explain your reasoning? Recall that your code, when executed, should transform the pregrid into the postgrid. Be careful with the AVATAR in the postgrid as it should show the effect of the code execution.

(b) Prompts used for obtaining the main part of $\mathtt{T}_{\mathrm{IO}}^{\mathrm{out}}$

Figure 24: Prompts used in the implementation of GPT4TaskSyn-*converse* technique for Karel domain.

Code: Few-shot prompt

I am working in the block-based visual programming domain of Hour of Code: Maze Challenge from code.org. In this domain, the following types of coding blocks are available:

- Basic action blocks: move forward, turn left, turn right.

- Boolean conditions: path ahead, path left, path right.

- Loops: repeatUntil(goal), repeat(int).

- Conditionals: if(boolean), if(boolean)else.

In this domain, a task is represented as an 8x8 visual grid that contains WALL cells, FREE cells, AVATAR (with specific location and direction), and GOAL. We represent a task's 8x8 visual grid with the following symbols.

represents a WALL cell.

+ represents a FREE cell.

* represents GOAL.

E represents AVATAR's location facing East direction.

W represents AVATAR's location facing West direction.

N represents AVATAR's location facing North direction.

S represents AVATAR's location facing South direction.

Below, I will give you a program structure. Can you generate a code that respects this program structure?

You should not change the structure. This means that you shouldn't add or remove any loops (e.g., repeatUntil(goal), repeat(int)) and conditionals (e.g., if(boolean), if(boolean)else). The program needs to be valid, meaning that bodies of constructs cannot remain empty. To complete this given structure, you can use basic action blocks, boolean conditions, and iteration numbers that are available in the Hour of Code: Maze Challenge programming.

I am giving you some examples comprising a program structure and a code that respects the structure. Provide the code for the last structure.

— Example i: Structure —

[SKETCH_i]

— Example i: Code —

[CODE_i]

— Example n: Structure —

[SKETCH_n]

— Example n: Code —

(a) Prompts used for obtaining $\mathtt{C}^{\text{out}}$

Task: Few-shot prompt

I am working in the block-based visual programming domain of Hour of Code: Maze Challenge from code.org. In this domain, the following types of coding blocks are available.

- Basic action blocks: move forward, turn left, turn right.

- Boolean conditions: path ahead, path left, path right.

- Loops: repeatUntil(goal), repeat(int).

- Conditionals: if(boolean), if(boolean)else.

In this domain, a task is represented as an 8x8 visual grid that contains WALL cells, FREE cells, AVATAR (with specific location and direction), and GOAL. We represent a task's 8x8 visual grid with the following symbols.

represents a WALL cell.

+ represents a FREE cell.

* represents GOAL.

E represents AVATAR's location facing East direction.

W represents AVATAR's location facing West direction.

N represents AVATAR's location facing North direction.

S represents AVATAR's location facing South direction.

Below I will give you a solution code. Can you generate a task with 8x8 visual grid that would be solved by this code?

The visual grid must contain AVATAR (with specific location and direction) along with GOAL, and can have WALL cells and FREE cells. Number your grid with row numbers (1 to 8) and column numbers (1 to 8). Also, you should tell me the position of AVATAR and GOAL in your generated task so we are sure about the numbering.

You can verify the correctness of your generated task by executing the solution code on your task. A solution code for a task takes AVATAR to GOAL when executed. Note that AVATAR can only move on FREE cells and will crash if it tries to go to a WALL cell. If your generated task is not correct, you should try again to generate a correct task.

I am giving you some examples comprising a solution code and task that is solved by this code. Provide the task for the last solution code.

— Example i: Solution —

[CODE_i]

— Example i: Task —

[TASK_i]

— Example n: Solution —

[CODE_n]

— Example n: Task —

(b) Prompts used for obtaining the main part of $\mathtt{T}_{\mathrm{IO}}^{\mathrm{out}}$

Figure 25: Prompts used in the implementation of GPT4TaskSyn-*fewshot* technique for HoCMaze domain.

Code: Few-shot prompt

I am working in the block-based visual programming domain of Karel programming. In this domain, the following types of coding blocks are available:

- Basic action blocks: move forward, turn left, turn right, pick marker, put marker.

- Boolean conditions: path ahead, path left, path right, marker present, no path ahead, no marker present.

- Loops: while(boolean), repeat(int).

- Conditionals: if(boolean), if(boolean)else.

In this domain, a task is represented as a pair of 10x10 visual pregrid and 10x10 visual postgrid. This pregrid and postgrid contain WALL cells, FREE cells, AVATAR (with specific location and direction), and markers. We represent a task's 10x10 visual pregrid and postgrid with the following symbols.

represents a WALL cell.

+ represents a FREE cell.

m represents a cell with marker.

E represents AVATAR's location on a cell without marker, facing East direction.

W represents AVATAR's location on a cell without marker, facing West direction.

N represents AVATAR's location on a cell without marker, facing North direction.

S represents AVATAR's location on a cell without marker, facing South direction.

Em represents AVATAR's location on a cell with marker, facing East direction.

Wm represents AVATAR's location on a cell with marker, facing West direction.

Nm represents AVATAR's location on a cell with marker, facing North direction.

Sm represents AVATAR's location on a cell with marker, facing South direction.

Below, I will give you a program structure. Can you generate a code that respects this program structure?

You should not change the structure. This means that you shouldn't add or remove any loops (e.g., while(boolean), repeat(int)) and conditionals (e.g., if(boolean), if(boolean)else). The program needs to be valid, meaning that bodies of constructs cannot remain empty. To complete this given structure, you can use basic action blocks, boolean conditions, and iteration numbers that are available in Karel programming.

I am giving you some examples comprising a program structure and a code that respects the structure. Provide the code for the last structure.

— Example i: Structure —

[SKETCH_i]

— Example i: Code —

[CODE_i]

— Example n: Structure —

[SKETCH_n]

— Example n: Code —

(a) Prompts used for obtaining $\mathtt{C}^{\text{out}}$

Task: Few-shot prompt

I am working in the block-based visual programming domain of Karel programming. In this domain, the following types of coding blocks are available.

- Basic action blocks: move forward, turn left, turn right, pick marker, put marker.

- Boolean conditions: path ahead, path left, path right, marker present, no path ahead, no marker present.

- Loops: while(boolean), repeat(int).

- Conditionals: if(boolean), if(boolean)else.

In this domain, a task is represented as a pair of 10x10 visual pregrid and 10x10 visual postgrid. This pregrid and postgrid contain WALL cells, FREE cells, AVATAR (with specific location and direction), and markers. We represent a task's 10x10 visual pregrid and postgrid with the following symbols.

represents a WALL cell.

+ represents a FREE cell.

m represents a cell with marker.

E represents AVATAR's location on a cell without marker, facing East direction.

W represents AVATAR's location on a cell without marker, facing West direction.

N represents AVATAR's location on a cell without marker, facing North direction.

S represents AVATAR's location on a cell without marker, facing South direction.

Em represents AVATAR's location on a cell with marker, facing East direction.

Wm represents AVATAR's location on a cell with marker, facing West direction.

Nm represents AVATAR's location on a cell with marker, facing North direction.

Sm represents AVATAR's location on a cell with marker, facing South direction.

Below I will give you a solution code. Can you generate a task with a pair of 10x10 visual pregrid and 10x10 visual postgrid that would be solved by this code?

Both the visual pregrid and visual postgrid must contain AVATAR (with specific location and direction), and can have WALL cells, FREE cells, and markers. Number your grids with row numbers (1 to 10) and column numbers (1 to 10). Also, you should tell me the position of AVATAR in your generated pregrid and postgrid so we are sure about the numbering.

You can verify the correctness of your generated task by executing the solution code on your task. A solution code for a task transforms the pregrid into the postgrid when executed. Note that AVATAR can only move on FREE cells and will crash if it tries to go to a WALL cell. If your generated task is not correct, you should try again to generate a correct task.

I am giving you some examples comprising a solution code and task that is solved by this code. Provide the task for the last solution code.

— Example i: Solution —

[CODE_i]

— Example i: Task Pregrid —

[TASK_PREGRID_i]

– Example i: Task Postgrid —

[TASK_POSTGRID_i]

— Example n: Solution —

[CODE_n]

— Example n: Task Pregrid —

(b) Prompts used for obtaining the main part of $\mathtt{T}_{\mathrm{IO}}^{\mathrm{out}}$

Figure 26: Prompts used in the implementation of GPT4TaskSyn-*fewshot* technique for Karel domain.