# Adaptive Scaffolding in Block-Based Programming via Synthesizing New Tasks as Pop Quizzes

**Ahana Ghosh**[1] **Sebastian Tschiatschek**[2] **Sam Devlin**[3] **Adish Singla**[1]
[1]MPI-SWS, {gahana, adishs}@mpi-sws.org
[2]University of Vienna, sebastian.tschiatschek@univie.ac.at
[3]Microsoft Research, sam.devlin@microsoft.com

## Abstract

Block-based programming environments are increasingly used to introduce computing concepts to beginners. However, novice students often struggle in these environments, given the conceptual and open-ended nature of programming tasks. To effectively support a student struggling to solve a given task, it is important to provide adaptive scaffolding that guides the student towards a solution. We introduce a scaffolding framework based on pop quizzes presented as multi-choice programming tasks. To automatically generate these pop quizzes, we propose a novel algorithm, PQUIZSYN. More formally, given a reference task with a solution code and the student's current attempt, PQUIZSYN synthesizes new tasks for pop quizzes with the following features: (a) *Adaptive* (i.e., individualized to the student's current attempt), (b) *Comprehensible* (i.e., easy to comprehend and solve), and (c) *Concealing* (i.e., do not reveal the solution code). Our algorithm synthesizes these tasks using techniques based on symbolic reasoning and graph-based code representations. We show that our algorithm can generate hundreds of pop quizzes for different student attempts on reference tasks from *Hour of Code: Maze Challenge* [11] and *Karel* [9]. We assess the quality of these pop quizzes through expert ratings using an evaluation rubric. Further, we have built an online platform for practicing block-based programming tasks empowered via pop quiz based feedback, and report results from an initial user study.

## 1 Introduction

The emergence of block-based visual programming platforms has made coding more interactive and appealing for novice students. Block-based programming uses "code blocks" that reduce the burden of syntax and focuses on key programming concepts. Led by the success of languages like *Scratch* [33], initiatives like *Hour of Code* by Code.org [12], and online courses like *Intro to Programming with Karel* by CodeHS.com [9, 25], block-based programming has become integral to introductory CS education.

Programming tasks on these platforms are conceptual and open-ended, requiring multi-step deductive reasoning to solve, thereby making them challenging for students. To effectively support a struggling student to solve a particular task, it is important to provide feedback on their attempts. However, on platforms that have millions of students, it is infeasible for human tutors to provide feedback. Hence, there is a critical need for automated feedback generation systems to provide personalized support to students [22, 13]. Existing work in the domain has explored various methods of personalized feedback generation within a task, such as providing next-step hints in the form of next code blocks to use in a student attempt [35, 26, 36, 44, 31], providing adaptive worked examples [32, 43, 28], and providing data-driven analysis of a student's misconceptions [6, 40, 39, 41, 24, 16, 5].

Preprint. Under review.

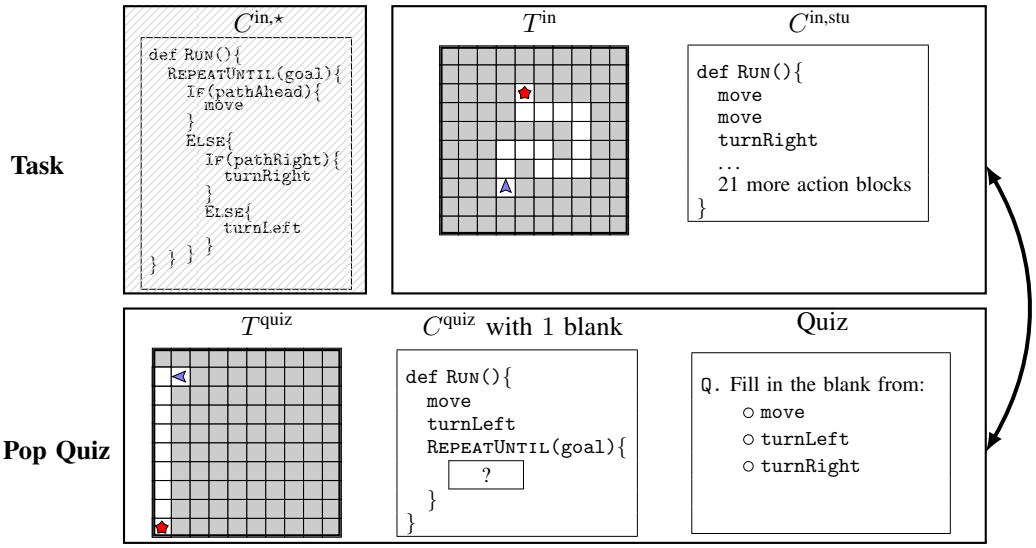



Figure 1: Illustration of our pop quiz based framework. The "Task" panel shows an input task $T^{\text{in}}$ from HOC [11], the student's current attempt $C^{\text{in,stu}}$, and the solution code $C^{\text{in},\star}$ (not revealed to the student). The student is currently unsuccessful in solving the task: the current attempt $C^{\text{in,stu}}$ does not solve the visual puzzle within the maximal number of permitted blocks (7 blocks) and does not use any of the required constructs (RepeatUntil and IfElse constructs). The "Pop Quiz" panel shows a pop quiz generated by our algorithm in the form of task-code pair ($T^{\text{quiz}}, C^{\text{quiz}}$) along with a multiple choice question, introducing the RepeatUntil construct. After the student solves the pop quiz, they resume working on the input task. The framework would be invoked when a student needs help; importantly, the pop quizzes presented to the student are adaptive w.r.t. the student's current attempt $C^{\text{in,stu}}$. Moreover, our algorithm generates pop quizzes that are easy to comprehend and solve, and $C^{\text{quiz}}$ sufficiently conceals $C^{\text{in},\star}$.

In this paper, we investigate an alternate method of personalized feedback generation that guides a student towards a task's solution while involving inquiry-driven and problem-solving aspects [14]. In particular, we introduce a scaffolding framework based on pop quizzes that contain new programming tasks presented as multi-choice questions.[1] Our framework is inspired by prior studies that showed the efficacy of multi-choice questions in helping novice students learn to code [30, 42, 19, 38, 17]. The framework is designed to be invoked as follows: Given a task and a student's current unsuccessful attempt, the framework can help the student by presenting a pop quiz intended to resolve their misconception. For the scaffolding to be effective, we center the design of the new programming task for a pop quiz around three features: *Adaptive*, *Comprehensible*, and *Concealing*; see details in Fig. 1 and Section 2.1. However, hand-crafting these new quizzes is time-consuming and potentially error-prone when required for a large number of tasks and different student attempts. To this end, we seek to *automatically* generate these pop quizzes by synthesizing new programming tasks.

### 1.1 Key Challenges and Our Contributions

There are several challenges in synthesizing new visual programming tasks with the above mentioned features, including the following: (i) current techniques for synthesizing visual programming tasks do not adapt to student attempts [1]; (ii) the mapping from the space of visual tasks to their solution codes is highly discontinuous as shown in [1], and hence task mutation based techniques are ineffective [37, 27]; (iii) the space of possible tasks and their solutions is potentially unbounded, and hence techniques that rely on exhaustive enumeration are intractable [37, 2, 4].

In this work, we develop a novel algorithm, PQuizSyn, that synthesizes pop quizzes with the desirable features of our scaffolding framework. Our algorithm overcomes the above-mentioned

[1] We refer to these multi-choice questions as "pop quizzes" as the framework could present these quizzes whenever a student needs help [7].

challenges by using techniques of symbolic execution, search algorithms, and graph-based code representations. Our key contributions are: **(I)** We present a modular and extensible algorithm for generating pop quizzes that operates in three stages (see Sections 2 and 3);[2] **(II)** We show that our approach can generate hundreds of pop quizzes for different types of student attempts on reference tasks from real-world programming platforms (see Section 4); **(III)** We assess the quality of our algorithm through expert ratings using a multi-dimensional evaluation rubric (see Section 5); **(IV)** We have built an online platform with our framework and demonstrate the utility of pop quiz based feedback through an initial user study (see Section 6).[3]

### 1.2 Additional Related Work

**Feedback via modelling programming concepts.** Apart from the above-mentioned methods such as next-step hints, there has been extensive work on feedback generation via modelling programming concepts. Here, several techniques have been proposed, including: (a) detecting challenging concepts by analyzing student attempts [6, 40, 39]; (b) discovering student misconceptions using task-specific rubrics and neural program embeddings [41]; (c) defining concepts through knowledge components [34, 15, 3].

**Evaluation of feedback methods.** An important aspect to consider when developing feedback generation methods is their evaluation criteria. Most next-step feedback generation methods are evaluated based on expert annotations or automated procedures [26, 24, 29]. In contrast, example-driven feedback techniques are typically evaluated using a multi-dimensional rubric [32, 43]. In our work, we evaluate the scaffolding framework through expert ratings using a rubric, as well as an initial user study.

## 2 Problem Setup and Definitions

In this section, we formalize our objective and introduce important technical definitions.

### 2.1 Problem Setup

**Task space.** We define the space of tasks as $\mathbb{T}$. A task $T \in \mathbb{T}$ consists of a visual puzzle and a set of available types of code blocks (e.g., `move`, `REPEATUNTIL`) allowed in the solution code. Additionally, the solution code must be within a certain size threshold in terms of the number of code blocks. We denote the current task that a student is solving as $T^{\text{in}} \in \mathbb{T}$; see $T^{\text{in}}$ in Fig. 1. In this work, we use tasks from *Hour of Code: Maze Challenge* [11] by Code.org [10] and *Intro to Programming with Karel* [9] by CodeHS.com [8]; henceforth, we refer to them as HOC and Karel tasks, respectively.

**Code space.** We define the space of all possible codes as $\mathbb{C}$ and represent them using a *Domain Specific Language* (DSL) [20]. In particular, for codes relevant for HOC and Karel tasks, we use a DSL based on [1]. A code $C \in \mathbb{C}$ has the following attributes: $C_{\text{blocks}}$ is the set of types of code blocks used in $C$, $C_{\text{size}}$ is the number of blocks used, and $C_{\text{depth}}$ is the depth of the *Abstract Syntax Tree* of $C$. We denote a distance metric in this space as $D_{\mathbb{C}}$. For a given $C \in \mathbb{C}$ and a positive integer $l$, we define a neighborhood function as $\mathcal{N}_{\mathbb{C}}(C, l) = \{C' \mid D_{\mathbb{C}}(C', C) \leq l\}$. The solution code $C^{\text{in},\star} \in \mathbb{C}$ for the task $T^{\text{in}}$ solves the visual puzzle using the allowed types of code blocks within the specified size threshold. A student attempt for $T^{\text{in}}$ is denoted as $C^{\text{in,stu}} \in \mathbb{C}$.

**Objective.** For an input task $T^{\text{in}}$ with solution code $C^{\text{in},\star}$ and given the current student attempt $C^{\text{in,stu}}$, our objective is to generate a pop quiz in form of a new task-code pair $(T^{\text{quiz}}, C^{\text{quiz}})$ designed on the basis of the following features: (i) *Adaptive*, i.e., $C^{\text{quiz}}$ accounts for $C^{\text{in},\star}$ and $C^{\text{in,stu}}$, ensuring that $C^{\text{quiz}}$ is individualized to the student's current attempt; (ii) *Comprehensible*, i.e., $C^{\text{quiz}}$ solves $T^{\text{quiz}}$ correctly and the pop quiz is easy to comprehend/solve without confusing the student; (iii) *Concealing*, i.e., $D_{\mathbb{C}}(C^{\text{quiz}}, C^{\text{in},\star})$ is high, ensuring that $C^{\text{quiz}}$ sufficiently conceals the solution code $C^{\text{in},\star}$ and does not directly reveal it in order to encourage problem-solving aspects.

[2] Implementation of the algorithm is publicly available at `https://github.com/machine-teaching-group/aied2022_pquizsyn_code`

[3] Online platform is publicly available at `https://www.teaching-blocks-hints.cc/`

### 2.2 Technical Definitions

**Sketch space.** We capture the key conceptual elements of a code using a higher level abstraction called a *sketch* [37, 2]. The sketch of a code preserves its important programming constructs. Similar to the code DSL, we define the sketch space $\mathbb{S}$ using a sketch DSL based on [1]. Similar to the *Abstract Syntax Tree* representation of a code, we represent a sketch as a tree having the programming constructs as its nodes. The mapping from the code space to the sketch space is captured by the many-to-one map, $\Psi : \mathbb{C} \to \mathbb{S}$, i.e., the representation of a code $C$ in $\mathbb{S}$ is given by $\Psi(C)$. As $\mathbb{S}$ is an abstraction of $\mathbb{C}$, multiple elements of $\mathbb{C}$ can correspond to a single element in $\mathbb{S}$. Similar to $D_{\mathbb{C}}$ and $\mathcal{N}_{\mathbb{C}}$, we denote a distance metric in the sketch space as $D_{\mathbb{S}}$ and a neighborhood function as $\mathcal{N}_{\mathbb{S}}(S, l) = \{S' \mid D_{\mathbb{S}}(S', S) \leq l\}$ for a given $S \in \mathbb{S}$ and a positive integer $l$.

**Sketch substructures.** For a sketch $S$, we define a substructure as a sub-tree containing the nodes of $S$ up to a particular depth and sharing the same root node; note that a substructure of a sketch is also a sketch. We denote the set of all substructures of $S$ as $\textsc{SubStructs}(S) \subseteq \mathbb{S}$; the size of the set $\textsc{SubStructs}(S)$ is typically small. For example, the sketch shown in Fig. 2b has the following 4 substructures: (i) {Run}, (ii) {Run {RepeatUntil(goal)}}, (iii) {Run {RepeatUntil(goal){IfElse (B)}}}, and (iv) {Run {RepeatUntil(goal){IfElse (B){{}; {IfElse (B)}}}}}.

**Code reductions.** For a code $C \in \mathbb{C}$ with sketch $S := \Psi(C)$, consider one of the sketches $S_{\text{sub}} \in \textsc{SubStructs}(S)$. We define the set of code reductions of $C$ w.r.t. sketch $S_{\text{sub}}$ as all codes obtained by removing one or more nodes of $C$ while preserving the sketch $S_{\text{sub}}$; note that the reduction of a code is also a code. We denote the set of all reductions as $\textsc{RedCodes}(C \mid S_{\text{sub}}) \subseteq \mathbb{C}$. For example, for $C^{\text{in},\star}$ in Fig. 1 and $S_{\text{sub}} = \{\texttt{Run}\{\texttt{RepeatUntil(goal)}\}\}$, the set $\textsc{RedCodes}(C^{\text{in},\star} \mid S_{\text{sub}})$ has the following 3 codes: (i) {Run {RepeatUntil(goal){move}}}, (ii) {Run{RepeatUntil(goal){turnRight}}}, and (iii) {Run{RepeatUntil(goal){turnLeft}}}.

## 3 Our Algorithm PQuizSyn

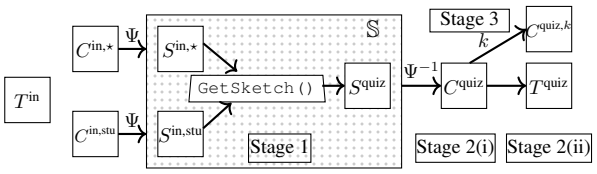
In this section, we present our algorithm that generates pop quizzes via synthesizing new tasks. One might be tempted to synthesize tasks by first generating a new visual puzzle and then obtaining its solution code. As discussed in Section 1 and shown in [1], the mapping from the space of visual tasks to their solution codes is highly discontinuous and reasoning about desirable tasks directly in the task space is ineffective. However, the task synthesis algorithm from [1] is not applicable to our work as we seek to generate tasks that also account for the student's current attempt. To this end, we develop a novel algorithm PQuizSyn (*Programming Pop Quizzes via Synthesis*) that generates tasks adaptive to the student's current attempt. Our algorithm operates in three stages: (i) Stage 1 generates a sketch based on the task's solution code and the student's current attempt; (ii) Stage 2 instantiates this sketch in the form of a new task-code pair; (iii) Stage 3 generates the pop quiz from the new task-code pair. Fig. 2a illustrates these stages, and details are provided below.

### 3.1 Stage 1: Generating the Pop Quiz Sketch $S^{\text{quiz}}$

We begin by describing Stage 1 of our algorithm as illustrated in Fig. 2a. In this stage, `GetSketch()` routine returns a suitable sketch $S^{\text{quiz}}$ that is instantiated in the later stages. The input to the routine is the student sketch $S^{\text{in,stu}} := \Psi(C^{\text{in,stu}})$ and solution sketch $S^{\text{in},\star} := \Psi(C^{\text{in},\star})$. By operating on the sketch space first, we can generate meaningful and adaptive codes in the later stages. To generate pop quizzes based on the features mentioned in Section 2.1, we require the sketch of the pop quiz $S^{\text{quiz}}$ to have the following attributes: (i) $S^{\text{quiz}}$ should direct the student towards the solution sketch $S^{\text{in},\star}$, i.e., $D_{\mathbb{S}}(S^{\text{quiz}}, S^{\text{in},\star})$ should be low; (ii) $S^{\text{quiz}}$ should be adaptive w.r.t. the student's sketch $S^{\text{in,stu}}$, i.e., $S^{\text{quiz}} \in \mathcal{N}_{\mathbb{S}}(S^{\text{in,stu}}, l)$ for a low value of $l$. While these conditions ensure that $S^{\text{quiz}}$ directs the student towards the solution sketch and is adaptive, it could potentially lead to a sketch that does not belong to the set of substructures of the solution sketch, i.e., $S^{\text{quiz}} \notin \textsc{SubStructs}(S^{\text{in},\star})$—in that case, there is no valid code reduction of $C^{\text{in},\star}$ w.r.t. $S^{\text{quiz}}$ (see Section 2.2) and this makes it challenging to instantiate sketches into desirable codes $C^{\text{quiz}}$ (see algorithm variant PQS-OneHop in Section 5 and Footnote 4). Hence, `GetSketch()` generates $S^{\text{quiz}}$ as follows (see Fig. 3):

(i) Pick $\hat{l}$ as $\min l \in \{1, 2, \ldots\}$ s.t. $\mathcal{N}_{\mathbb{S}}(S^{\text{in,stu}}, l) \cap \textsc{SubStructs}(S^{\text{in},\star})$ is non-empty.

(ii) Generate $S^{\text{quiz}} \in \operatorname{argmin}_{S \in \mathcal{N}_{\mathbb{S}}(S^{\text{in,stu}}, \hat{l}) \cap \textsc{SubStructs}(S^{\text{in},\star})} D_{\mathbb{S}}(S, S^{\text{in},\star})$.




(a) Our algorithm PQUIZSYN

```
def Run(){
 RepeatUntil(goal){
    If(B){}
    Else{
      If(B){}
      Else{}
    }
 }
}
```

(b) $S^{\text{in},\star}$

```
def Run(){}
```

(c) $S^{\text{in,stu}}$

```
def Run(){
 RepeatUntil(goal){}
}
```

(d) $S^{\text{quiz}}$

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

(e) $C^{\text{quiz}}$

Figure 2: (a) illustrates PQUIZSYN. In particular, we can instantiate the presented algorithm using input task $T^{\text{in}}$, its solution code $C^{\text{in},\star}$, and the current student attempt $C^{\text{in,stu}}$ from Fig. 1. The sketch of $C^{\text{in},\star}$ is shown in (b), sketch of $C^{\text{in,stu}}$ is shown in (c), sketch of $C^{\text{quiz}}$ is shown in (d), and the code of the pop quiz $C^{\text{quiz}}$ is shown in (e).

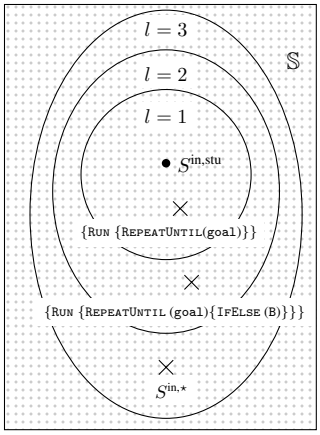



Figure 3: PQUIZSYN Stage 1 for the scenario shown in Fig. 1. X shows substructures of $S^{\text{in},\star}$ in $l$-hop neighborhoods of $S^{\text{in,stu}}$ for $l \in \{1, 2, 3\}$. Details are provided in Section 3.1.

### 3.2 Stage 2: Synthesizing $(T^{\text{quiz}}, C^{\text{quiz}})$ from $S^{\text{quiz}}$

Next, we describe Stage 2 of our algorithm. We first generate $C^{\text{quiz}}$ from $S^{\text{quiz}}$, as illustrated in Stage 2(i) of Fig. 2a. Specifically, for a sketch $S^{\text{quiz}}$ generated in Stage 1, we employ the *code mutation* methodology proposed in [1] to obtain a code $C^{\text{quiz}}$. However, this methodology requires a meaningful starting code $C^{\text{seed}}$. Since $S^{\text{quiz}} \in \text{SUBSTRUCTS}(S^{\text{in},\star})$ by the design of Stage 1, we begin by picking $C^{\text{seed}}$ from the set $\text{REDCODES}(C^{\text{in},\star} |\ S^{\text{quiz}})$.[4] The methodology of [1] provides us multiple code mutations of $C^{\text{seed}}$. The extent to which these code mutations differ from $C^{\text{seed}}$ and $C^{\text{in},\star}$ is controlled by the constraints imposed based on the values of the boolean variables, conditionals, and action blocks (`move`, `turnLeft`, `turnRight`, `pickMarker`, `putMarker`) of $C^{\text{seed}}$, as well as constraints on the size of the obtained code. Specifically, these mutations allow us to control the extent to which $D_{\mathbb{C}}(C^{\text{quiz}}, C^{\text{in},\star})$ varies, which is a desired feature as stated in Section 2.1.

Next, we generate a new task $T^{\text{quiz}}$ from a code $C^{\text{quiz}}$ as illustrated in Stage 2(ii) of Fig. 2a. Specifically, we generate $T^{\text{quiz}}$ such that its solution code is $C^{\text{quiz}}$. We achieve this using techniques of symbolic execution and best-first search, building on the task synthesis methodology presented in [1].

### 3.3 Stage 3: Generating Multi-Choice Question from $(T^{\text{quiz}}, C^{\text{quiz}})$

In this stage, we generate a pop quiz with a fixed set of answer choices; see Figs. 1 and 5. We pick a task-code pair $(T^{\text{quiz}}, C^{\text{quiz}})$, and expose only a part of $C^{\text{quiz}}$ determined by an exposure parameter $k$, i.e., $C^{\text{quiz}}$ contains $k$ blanks. These blanks must be filled out by the student from the set of answer choices in a manner that would solve $T^{\text{quiz}}$. Specifically, we generate the pop quiz with $k = 1$ blanks. To obtain the blank for the quiz, we do an in-order traversal of $C^{\text{quiz}}$ and leave out the last leaf node as blank.

## 4 PQUIZSYN on Real-World Tasks

In this section, we present the performance of PQUIZSYN on six reference tasks taken from real-world block-based programming platforms: HOC [11] and Karel [9]. The set of these tasks along with their sources are mentioned in Fig. 4. These tasks differ in complexity, measured in terms of the programming constructs of their solution code as illustrated by the diversity of their respective solution sketches $S^{\text{in},\star}$. For the exhaustive set of substructures of $S^{\text{in},\star}$, Fig. 4 lists the total number

[4] When $S^{\text{quiz}} \notin \text{SUBSTRUCTS}(S^{\text{in},\star})$, we set $C^{\text{seed}}$ as a random instantiation of $S^{\text{quiz}}$ – see algorithm variant PQS-ONEHOP in Section 5.

| Name, source for $T^{\text{in}}$ | $C^{\text{in},\star}_{\text{size}}$, $S^{\text{in},\star}$ for $T^{\text{in}}$ | $S^{\text{quiz}} \in \textsc{SubStructs}(S^{\text{in},\star})$ | $\#C^{\text{quiz}}$ | $\#T^{\text{quiz}}$ |
|---|---|---|---|---|
| T-1 **HOC:Maze08** [11] | 6 {Run {Repeat; Repeat}} | {Run}<br>{Run {Repeat}}<br>$S^{\text{in},\star}$ | 22<br>34<br>179 | 220<br>340<br>1790 |
| T-2 **HOC:Maze16** [11] | 5 {Run {RUntil {If}}} | {Run}<br>{Run {RUntil}}<br>$S^{\text{in},\star}$ | 10<br>6<br>19 | 100<br>60<br>190 |
| T-3 **HOC:Maze18** [11] | 5 {Run {RUntil {IfElse}} | {Run}<br>{Run {RUntil}}<br>$S^{\text{in},\star}$ | 10<br>6<br>9 | 100<br>60<br>90 |
| T-4 **HOC:Maze20** [11] | 7 {Run {RUntil {IfElse {{};{IfElse}}}}} | {Run}<br>{Run {RUntil}}<br>{Run {RUntil {IfElse}}}<br>$S^{\text{in},\star}$ | 10<br>6<br>9<br>10 | 100<br>60<br>90<br>100 |
| T-5 **Karel:Opposite** [9] | 6 {Run {Repeat {IfElse}}} | {Run}<br>{Run {Repeat}}<br>$S^{\text{in},\star}$ | 73<br>118<br>343 | 730<br>1180<br>3430 |
| T-6 **Karel:Diagonal** [9] | 8 {Run {While}} | {Run}<br>$S^{\text{in},\star}$ | 447<br>579 | 4470<br>5790 |

Figure 4: PQuizSyn applied to six HOC and Karel reference tasks; see Section 4 for details. For brevity, sketches have been abbreviated, e.g., `RepeatUntil(goal)` as `RUntil`.

of pop quizzes, in the form of unique task-code pairs ($T^{\text{quiz}}, C^{\text{quiz}}$), generated by our algorithm. As can be seen in the figure, our algorithm generates 50 to 1000s of pop quizzes for each substructure. For any potential student attempt on these tasks, Stage 1 of PQuizSyn would generate one of these task-specific substructures by design – hence, for every attempt we can present several unique yet adaptive pop quizzes to the student. Note that, our algorithm generates higher number of tasks than codes for each substructure. This is because the task synthesis methodology used in Stage 2(ii) can generate more than one task for a single code in Stage 2(ii) of Fig. 2a. In particular, for each new code, we obtain 10 diverse tasks. For instance, Fig. 1 and Fig. 5 illustrate pop quizzes generated by PQuizSyn for the specific student attempts on tasks T-4 and T-5, respectively.

## 5 Expert Study via Multi-Dimensional Rubric

In this section, we evaluate PQuizSyn w.r.t. the desired features specified in the objective, i.e., *Adaptive*, *Comprehensible*, and *Concealing* (see Section 2.1). In particular, we seek to compare PQuizSyn with its variants resulting from different design choices in Section 3. To this end, we conduct an expert study via a multi-dimensional rubric.

**Variants of PQuizSyn algorithm.** We compare the performance of PQuizSyn with the following variants: PQS-FullHop, PQS-OneHop, and PQS-RedCode. PQS-FullHop and PQS-OneHop differ from PQuizSyn only in the `GetSketch()` routine used in Stage 1 of Fig. 2a when generating $S^{\text{quiz}}$. In particular, Stage 1 of PQS-FullHop always returns the sketch of the solution code, i.e., $S^{\text{quiz}} := S^{\text{in},\star}$; Stage 1 of PQS-OneHop returns a sketch directly from the 1-hop neighborhood of $S^{\text{in,stu}}$, i.e., $S^{\text{quiz}} \in \mathcal{N}_{\mathbb{S}}(S^{\text{in,stu}}, 1)$. The third baseline, PQS-RedCode, differs from PQuizSyn only in Stage 2(i) of Fig. 2a when generating $C^{\text{quiz}}$ from $S^{\text{quiz}}$. In particular, Stage 2(i) of PQS-RedCode generates $C^{\text{quiz}}$ as a direct reduction of the solution code w.r.t. the sketch obtained in Stage 1, i.e., $C^{\text{quiz}} \in \textsc{RedCodes}(C^{\text{in},\star} \mid S^{\text{quiz}})$.

**Simulated student attempts.** For this expert evaluation, we simulated unsuccessful student attempts as seen in block-based programming domains [26]. In particular, for each reference task, we manually created four student attempts as follows: (a) Stu-A: $C^{\text{in,stu}}$ uses only action blocks, i.e., (`move`, `turnLeft`, `turnRight`, `pickMarker`, `putMarker`); (b) Stu-B: $C^{\text{in,stu}}$ uses a subset of programming constructs in $C^{\text{in},\star}$; (c) Stu-C: $C^{\text{in,stu}}$ is structurally the same as $C^{\text{in},\star}$, i.e., $S^{\text{in,stu}} = S^{\text{in},\star}$; (d) Stu-D: $C^{\text{in,stu}}$ has a structure more complex than $C^{\text{in},\star}$. These four types of attempts exhaustively cover all the scenarios that an algorithm might encounter when deployed (see Section 6).

**Multi-dimensional evaluation rubric.** Inspired by the evaluation rubric in [32, 43], we assess pop quizzes on a multi-dimensional rubric with three attributes, each rated on a three-point Likert scale (with higher scores being better). More concretely, we have: (i) *Adaptive* attribute measuring the degree of individualization of the pop quiz to the current student attempt (3: high; 2: medium;

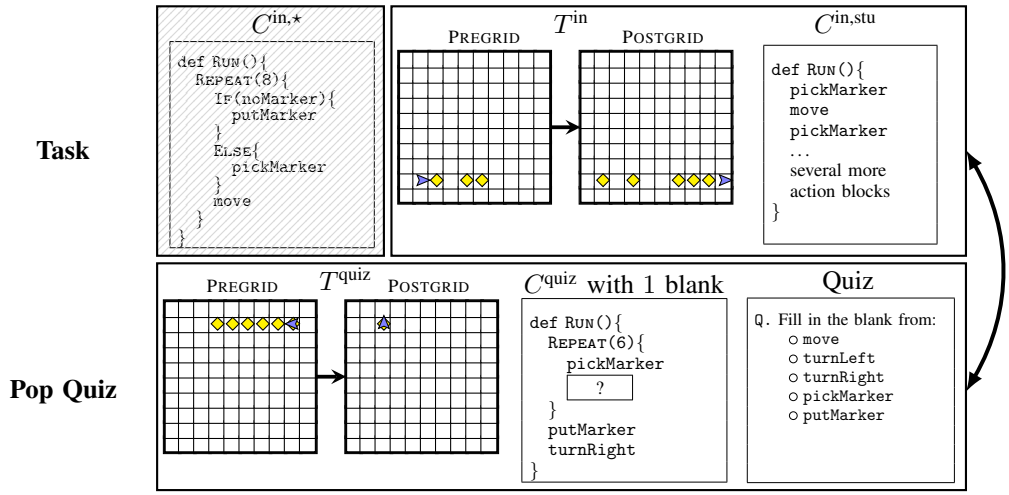



Figure 5: Analogous to Fig. 1, here we illustrate our framework on a Karel task, T-5 (see Fig. 4). Karel tasks [25] comprise of a pair of visual grids, (PREGRID, POSTGRID), and the objective is to write code that, when executed, transforms PREGRID to POSTGRID.

1: low); (ii) *Comprehensible* attribute measuring how easy the pop quiz is to comprehend/solve (3: easy; 2: might confuse the student sometimes; 1: either incorrect or is very difficult to solve.); (iii) *Concealing* attribute measuring the extent to which the pop quiz conceals the solution code (3: sufficiently conceals; 2: reveals the solution to some extent; 1: reveals the solution to a large extent). *Overall* denotes the sum of scores across three attributes for a pop quiz.

**Expert study setup.** We picked three tasks spanning different types of constructs and complexity: T-1, T-4, and T-5 from Fig. 4. Thus, in total we evaluated 48 scenarios: 4 algorithm variants $\times$ 4 student types $\times$ 3 tasks (see Figs. 1 and 5 as example scenarios). Two researchers, with experience in block-based programming, evaluated each of the 48 scenarios independently. The evaluation was done through a web survey where a scenario was introduced at random, and assessed based on the rubric.

**Expert study results.** First, we validate the expert ratings using the quadratic-weighted Cohen's kappa inter-agreement reliability value [32] for each attribute: 0.62 (*Adaptive*), 0.69 (*Comprehensible*), 0.79 (*Concealing*), and 0.7 (*Overall*). The values indicate *substantial agreement* between the raters. The average ratings are presented in Fig. 6 and PQUIZSYN has the highest *Overall* score. We analyze these ratings per attribute based on the Kruskal-Wallis significance test [21]; the results discussed next are statistically significant with $p < 0.01$. On the *Adaptive* attribute, PQS-FULLHOP performs significantly worse because it does not account for the student attempt (see Section 3.1). On the *Comprehensible* attribute, PQS-ONEHOP performs significantly worse because there are instances where no valid code reduction of $C^{in,\star}$ w.r.t. $S^{quiz}$ is found (see Footnote 4, Section 3.2). Finally, on the *Concealing* attribute, PQS-REDCODE performs significantly worse because it obtains $C^{quiz}$ via a direct reduction of $C^{in,\star}$ without any mutation (see Section 3.2).

| Algorithm | *Adaptive* | *Comprehensible* | *Concealing* | *Overall* |
|---|---|---|---|---|
| PQS-FULLHOP | 2.0(0.7) | 2.8(0.1) | 3.0(0.0) | 7.8(0.8) |
| PQS-ONEHOP | 2.8(0.1) | 2.5(0.6) | 3.0(0.0) | 8.3(0.7) |
| PQS-REDCODE | 2.7(0.3) | 3.0(0.0) | 1.5(0.4) | 7.2(0.7) |
| PQUIZSYN | 2.7(0.2) | 3.0(0.0) | 2.9(0.1) | 8.6(0.3) |

Figure 6: Mean (Variance) attribute ratings for different algorithms. Higher scores are better. PQUIZSYN performs well across all three attributes and has the highest *Overall* score; see Section 5 for details.

## 6 User Study via Online Platform

We have built an online platform with our PQUIZSYN framework using the Blockly Games library [18]. The online platform is publicly accessible – see Footnote 3, Section 1.1. The platform provides an interface for a participant to practice block-based programming tasks, and receive pop

quiz based feedback when stuck. In this section, we report results from an initial user study to assess the efficacy of our scaffolding framework in comparison to other feedback methods.

**Participation session and feedback methods.** A single session on our platform comprises of three steps. In STEP-A, the participant is presented with a task and has 10 execution tries to solve it. If a participant fails to solve the task at STEP-A, they proceed to STEP-B with a randomly assigned feedback method (NOHINT, NEXTSTEP, and PQUIZSYN as discussed below). After STEP-B, the participant resumes their attempt on the task in STEP-C with 10 additional execution tries. Note that the feedback method is invoked only once in a single session. Next, we describe different feedback methods at STEP-B. NOHINT represents a baseline where the participant is directed to STEP-C without any feedback. NEXTSTEP corresponds to next-step hints as feedback where the participant's code is updated to bring it closer to a solution code [35, 26, 36, 44, 31]; we prioritized next-step edits involving programming constructs (e.g., `REPEATUNTIL`) over basic actions (e.g., `move`). PQUIZSYN is our pop quiz based feedback.

**User study results.** We conducted an initial user study with participants recruited from Amazon Mechanical Turk; an IRB approval was received before the study. The participants were US-based adults, without expertise in block-based visual programming. Due to the costs involved (over 3 USD per participant), we selected two tasks for the study: T-3 and T-5 from Fig. 4. We present the detailed results in Fig. 7. In total, we had 575 unique participants; out of these, 0.774 fraction failed to solve the task at STEP-A and proceeded to STEP-B. PQUIZSYN was assigned to 148 participants in STEP-B (0.60 fraction successfully solved the presented pop quiz). Subsequently, 0.128 fraction of these participants solved the task in STEP-C. Here, 0.128 measures the success rate of participants assigned to PQUIZSYN; in comparison, it is 0.082 for NEXTSTEP and 0.046 for NOHINT – see Fig. 7. Overall, the performance of PQUIZSYN is better than NEXTSTEP (the gap is not significant w.r.t. $\chi^2$-test, $p = 0.19$) and NOHINT (the gap is significant w.r.t. $\chi^2$-test, $p = 0.01$) [23]. These initial results demonstrate the utility of providing pop quiz based feedback.

| Feedback | Total (**STEP-B**) | | | Fraction solved (**STEP-C**) | | |
|---|---|---|---|---|---|---|
| | Both | T-3 | T-5 | Both | T-3 | T-5 |
| NOHINT | 151 | 63 | 88 | 0.046 | 0.079 | 0.023 |
| NEXTSTEP | 146 | 63 | 83 | 0.082 | 0.127 | 0.048 |
| PQUIZSYN | 148 | 62 | 86 | 0.128 | 0.177 | 0.093 |

Figure 7: Results for tasks T-3 and T-5 ("Both" represents aggregated results). In **STEP-A**, we had a total of 575 (293 for T-3, 282 for T-5) participants; about 0.774 (0.642 for T-3, 0.911 for T-5) fraction failed to solve the task at **STEP-A** and proceeded to **STEP-B / STEP-C** with a randomly assigned feedback method.

## 7 Conclusions and Outlook

We proposed a novel scaffolding framework for block-based programming based on pop quizzes that involve inquiry-driven and problem-solving aspects. We developed a modular synthesis algorithm, PQUIZSYN, that generates these pop quizzes. After conducting an expert assessment using a multi-dimensional rubric, we developed an online platform empowered by our scaffolding framework. While initial user study results with our platform demonstrate the utility of our pop quiz based framework, there are several interesting directions to continue this study, including: (i) extending our platform to provide multiple rounds of feedback within a single participation session and measuring the efficacy of different methods; (ii) comparing our synthesized pop quizzes with those generated by experts; (iii) conducting longitudinal studies with novice students to measure long-term improvements in problem solving skills; (iv) extending our framework to more complex block-based programming domains.

**Acknowledgments.** We would like to thank the reviewers for their feedback. Ahana Ghosh was supported by Microsoft Research through its PhD Scholarship Programme. Adish Singla acknowledges support by the European Research Council (ERC) under the Horizon Europe programme (ERC StG, grant agreement No. 101039090).

## References


[1] 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.

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

[3] Bita Akram, Hamoon Azizsoltani, Wookhee Min, Eric N. Wiebe, Bradford W. Mott, Anam Navied, Kristy Elizabeth Boyer, and James C. Lester. Automated Assessment of Computer Science Competencies from Student Programs with Gaussian Process Regression. In *EDM*, 2020.

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

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

[6] Yuliya Cherenkova, Daniel Zingaro, and Andrew Petersen. Identifying Challenging CS1 Concepts in a Large Problem Dataset. In *SIGCSE*, 2014.

[7] Vincent A. Cicirello. On the Role and Effectiveness of Pop Quizzes in CS1. In *SIGCSE*, 2009.

[8] CodeHS.com. CodeHS – Teaching Coding and CS. `https://codehs.com/`.

[9] CodeHS.com. Intro to Programming with Karel the Dog. `https://codehs.com/info/curriculum/introkarel`.

[10] Code.org. Code.org – Learn Computer Science. `https://code.org/`.

[11] Code.org. Hour of Code – Classic Maze Challenge. `https://studio.code.org/s/hourofcode`.

[12] Code.org. Hour of Code Initiative. `https://hourofcode.com/`.

[13] Christa Cody, Mehak Maniktala, Nicholas Lytle, Min Chi, and Tiffany Barnes. The Impact of Looking Further Ahead: A Comparison of Two Data-driven Unsolicited Hint Types on Performance in an Intelligent Data-driven Logic Tutor. *IJAIED*, 2021.

[14] Lucas Cordova, Jeffrey C. Carver, Noah Gershmel, and Gursimran Walia. A Comparison of Inquiry-Based Conceptual Feedback vs. Traditional Detailed Feedback Mechanisms in Software Testing Education: An Empirical Investigation. In *SIGCSE*, 2021.

[15] Will Crichton, Georgia Gabriela Sampaio, and Pat Hanrahan. Automating Program Structure Classification. In *SIGCSE*, 2021.

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

[17] Alexandru Ene and Cosmin Stirbu. Automatic Generation of Quizzes for Java Programming Language. In *ECAI*, 2019.

[18] Blockly Games. Games for Tomorrow's Programmers. `https://blockly.games/`.

[19] Shuchi Grover. Toward A Framework for Formative Assessment of Conceptual Learning in K-12 Computer Science Classrooms. In *SIGCSE*, 2021.

[20] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program Synthesis. *Foundations and Trends® in Programming Languages*, 2017.

[21] Thomas W MacFarland and Jan M Yates. Kruskal–Wallis H-test for Oneway Analysis of Variance (ANOVA) by Ranks. In *Introduction to Nonparametric Statistics for the Biological Sciences using R*. Springer, 2016.

[22] Samiha Marwan, Ge Gao, Susan Fisk, Thomas Price, and Tiffany Barnes. Adaptive Immediate Feedback Can Improve Novice Programming Engagement and Intention to Persist in Computer Science. In *ICER*, 2020.

[23] Mary L McHugh. The Chi-Square Test of Independence. *Biochemia Medica*, 2013.

[24] Benjamin Paaßen, Barbara Hammer, Thomas William Price, Tiffany Barnes, Sebastian Gross, and Niels Pinkwart. The Continuous Hint Factory - Providing Hints in Continuous and Infinite Spaces. *JEDM*, 2018.

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

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

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

[28] Thomas W Price, Yihuan Dong, and Dragan Lipovac. isnap: Towards Intelligent Tutoring in Novice Programming Environments. In *SIGCSE*, 2017.

[29] Thomas W. Price, Yihuan Dong, Rui Zhi, Benjamin Paaßen, Nicholas Lytle, Veronica Cateté, and Tiffany Barnes. A Comparison of the Quality of Data-Driven Programming Hint Generation algorithms. *IJAIED*, 2019.

[30] Thomas W. Price, Joseph Jay Williams, Jaemarie Solyst, and Samiha Marwan. Engaging Students with Instructor Solutions in Online Programming Homework. In *CHI*, 2020.

[31] Thomas W. Price, Rui Zhi, and Tiffany Barnes. Evaluation of a Data-driven Feedback Algorithm for Open-ended Programming. In *EDM*, 2017.

[32] Thomas W. Price, Rui Zhi, and Tiffany Barnes. Hint Generation Under Uncertainty: The Effect of Hint Quality on Help-Seeking Behavior. In *AIED*, 2017.

[33] Mitchel Resnick et al. Scratch: Programming for All. *ACM*, 2009.

[34] Kelly Rivers, Erik Harpstead, and Kenneth R. Koedinger. Learning Curve Analysis for Programming: Which Concepts do Students Struggle With? In *ICER*, 2016.

[35] Kelly Rivers and Kenneth R. Koedinger. Automating Hint Generation with Solution Space Path Construction. In *ITS*, 2014.

[36] Kelly Rivers and Kenneth R. Koedinger. Data-Driven Hint Generation in Vast Solution Spaces: A Self-Improving Python Programming Tutor. *IJAIED*, 2017.

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

[38] Reza Soltanpoor, Charles Thevathayan, and Daryl J. D'Souza. Adaptive Remediation for Novice Programmers through Personalized Prescriptive Quizzes. In *ITiCSE*, 2018.

[39] Eliane Stampfer Wiese, Anna N. Rafferty, and Armando Fox. Linking Code Readability, Structure, and Comprehension Among Novices: It's Complicated. In *ICSE*, 2019.

[40] Eliane Stampfer Wiese, Anna N. Rafferty, Daniel M. Kopta, and Jacqulyn M. Anderson. Replicating Novices' Struggles with Coding Style. In *ICPC*, 2019.

[41] 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.

[42] Lishan Zhang, Baoping Li, Qiujie Zhang, and I. Hsiao. Does a Distributed Practice Strategy for Multiple Choice Questions Help Novices Learn Programming. *iJET*, 2020.

[43] Rui Zhi, Samiha Marwan, Yihuan Dong, Nicholas Lytle, Thomas W. Price, and Tiffany Barnes. Toward Data-Driven Example Feedback for Novice Programming. In *EDM*, 2019.

[44] Kurtis Zimmerman and Chandan Raj Rupakheti. An Automated Framework for Recommending Program Elements to Novices (N). In *ASE*, 2015.