From {Solution} Synthesis to {Student Attempt} Synthesis for Block-Based Visual Programming Tasks*

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

Nikitas Theodoropoulos
MPI-SWS
ntheodor@mpi-sws.org

ABSTRACT

Block-based visual programming environments are increasingly used to introduce computing concepts to beginners. Given that programming tasks are open-ended and conceptual, novice students often struggle when learning in these environments. AI-driven programming tutors hold great promise in automatically assisting struggling students, and need several components to realize this potential. We investigate the crucial component of student modeling, in particular, the ability to automatically infer students' misconceptions for predicting (synthesizing) their behavior. We introduce a novel benchmark, StudentSyn, centered around the following challenge: For a given student, synthesize the student's attempt on a new target task after observing the student's attempt on a fixed reference task. This challenge is akin to that of program synthesis; however, instead of synthesizing a {*solution*} (i.e., program an expert would write), the goal here is to synthesize a {*student attempt*} (i.e., program that a given student would write). We first show that human experts (TutorSS) can achieve high performance on the benchmark, whereas simple baselines perform poorly. Then, we develop two neuro/symbolic techniques (NeurSS and SymSS) in a quest to close this gap with TutorSS.

Keywords

block-based visual programming, programming education, program synthesis, neuro-symbolic AI, student modeling

1. INTRODUCTION

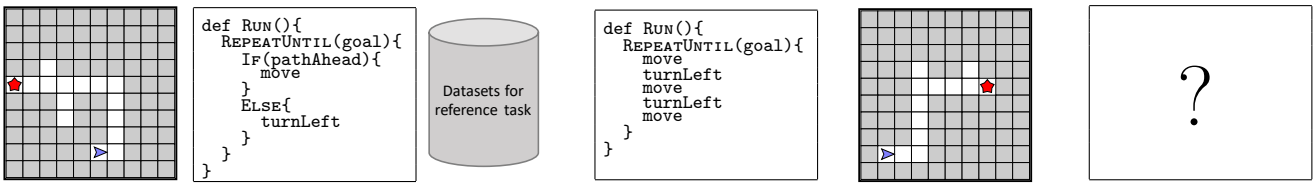
The emergence of block-based visual programming platforms has made coding more accessible and appealing to beginners. Block-based programming uses "code blocks" that reduce the burden of syntax and introduces concepts in an interactive way. Led by initiatives like *Hour of Code* by Code.org [10, 8] and the popularity of languages like *Scratch* [41], block-based programming has become integral to introductory CS education. Considering the *Hour of Code* initiative alone, over one billion hours of programming activity has been spent in learning to solve tasks in such environments [8].

Programming tasks on these platforms are conceptual and open-ended, and require multi-step deductive reasoning to solve. Given these aspects, novices often struggle when learning to solve these tasks. The difficulties faced by novice students become evident by looking at the trajectory of students' attempts who are struggling to solve a given task. For instance, in a dataset released by Code.org [10, 8, 35], even for simple tasks where solutions require only 5 code blocks (see Figure 1a), students submitted over $50,000$ unique attempts with some exceeding a size of 50 code blocks.

AI-driven programming tutors have the potential to support these struggling students by providing personalized assistance, e.g., feedback as hints or curriculum design [37]. To effectively assist struggling students, AI-driven systems need several components, a crucial one being *student modeling*. In particular, we need models that can automatically infer a student's knowledge from limited interactions and then predict the student's behavior on new tasks. However, student modeling in block-based visual programming environments can be quite challenging because of the following: (i) programming tasks are conceptual with no well-defined skill-set or problem-solving strategy for mastery [23]; (ii) there could be a huge variability in students' attempts for a task [52]; (iii) the objective of predicting a given student's behavior on new tasks is not limited to coarse-grained success/failure indicators (e.g., [50])—ideally, we should be able to do fine-grained synthesis of attempts for the student.

Beyond the above-mentioned challenges, there are two critical issues arising from limited resources and data scarcity for a given domain. First, while the space of tasks that could be designed for personalized curriculum is intractably large [1], the publicly available datasets of real-world students' attempts are limited; e.g., the *Hour of Code: Maze Challenge* domain has datasets for only two tasks [35]. Second, when a deployed system is interacting with a new student, there is limited prior information [15], and the system would have to infer the student's knowledge by observing behavior on a few reference tasks, e.g., through a quiz [21]. These two issues limit the applicability of state-of-the-art techniques that rely on large-scale datasets across tasks or personalized data per student (e.g., [50, 28, 29, 36])—we need next-generation student modeling techniques that can operate under data

*Correspondence to: Adish Singla <adishs@mpi-sws.org>; Authors listed alphabetically.



(a) Reference task $\mathtt{T}^{18}$ with solution code and datasets (b) `stu`'s attempt for $\mathtt{T}^{18}$ (c) Target task $\mathtt{T}^{18x}$ (d) `stu`'s attempt for $\mathtt{T}^{18x}$

Figure 1: Illustration of our problem setup and objective for the task Maze#18 in the *Hour of Code: Maze* [9] by Code.org [8]. As explained in Section 2.2, we consider three distinct phases in our problem setup to provide a conceptual separation in terms of information and computation available to a system. **(a)** In the first phase, we are given a reference task $\mathtt{T}^{18}$ along with its solution code $\mathtt{C}^{\star}_{\mathtt{T}^{18}}$ and data resources (e.g., a real-world dataset of different students' attempts); reference tasks are fixed and the system can use any computation a priori. **(b)** In the second phase, the system interacts with a student, namely `stu`, who attempts the reference task $\mathtt{T}^{18}$ and submits a code, denoted as $\mathtt{C}^{\mathtt{stu}}_{\mathtt{T}^{18}}$. **(c, d)** In the third phase, the system seeks to synthesize the student `stu`'s behavior on a target task $\mathtt{T}^{18x}$, i.e., a program that `stu` would write if the system would assign $\mathtt{T}^{18x}$ to the student. Importantly, the target task $\mathtt{T}^{18x}$ is not available a priori and this synthesis process would be done in real-time.

scarcity and limited observability. To this end, this paper focuses on the following question: *For a given student, can we synthesize the student's attempt on a new target task after observing the student's attempt on a fixed reference task?*

1.1 Our Approach and Contributions

Figure 1 illustrates this synthesis question for a scenario in the *Hour of Code: Maze Challenge* [9] by Code.org [8]. This question is akin to that of program synthesis [20]; however, instead of synthesizing a {*solution*} (i.e., program an expert would write), the goal here is to synthesize a {*student attempt*} (i.e., program that a given student would write). This goal of synthesizing student attempts, and not just solutions, requires going beyond state-of-the-art program synthesis techniques [3, 4, 25]; crucially, we also need to define appropriate metrics to quantitatively measure the performance of different techniques. Our main contributions are:

(1) We formalize the problem of synthesizing a student's attempt on target tasks after observing the student's behavior on a fixed reference task. We introduce a novel benchmark, StudentSyn, centered around the above synthesis question, along with generative/discriminative performance measures for evaluation.

(2) We showcase that human experts (TutorSS) can achieve high performance on StudentSyn, whereas simple baselines perform poorly.

(3) We develop two techniques inspired by neural (NeurSS) and symbolic (SymSS) methods, in a quest to close the gap with human experts (TutorSS).

We provide additional details and results in the longer version of the paper [47]. We will also publicly release the benchmark and implementations to facilitate future research.

1.2 Related Work

Student modeling. For close-ended domains like vocabulary learning ([42, 36, 22]) and Algebra problems ([12, 40, 43]), the skills or knowledge components for mastery are typically well-defined and we can use *Knowledge Tracing* techniques to model a student's knowledge state over time [11, 33]. These modeling techniques, in turn, allow us to provide feedback, predict solution strategies, or infer/quiz a student's knowledge state [40, 21, 43]. Open-ended domains pose unique challenges to directly apply these techniques (see [23]); however, there has been some progress in this direction. In recent works [28, 29], models have been proposed to predict human behavior in chess for specific skill levels and to recognize the behavior of individual players. Along these lines, [7] introduced methods to perform early prediction of struggling students in open-ended interactive simulations. There has also been work on student modeling for block-based programming, e.g., clustering-based methods for misconception discovery [18, 44], and deep learning methods to represent knowledge and predict performance [50].

AI-driven systems for programming education. There has been a surge of interest in developing AI-driven systems for programming education, and in particular, for block-based programming domains [37, 38, 51]. Existing works have studied various aspects of intelligent feedback, for instance, providing next-step hints when a student is stuck [35, 53, 31, 15], giving data-driven feedback about a student's misconceptions [45, 34, 39, 52], or generating/recommending new tasks [2, 1, 19]. Depending on the availability of datasets and resources, different techniques are employed: using historical datasets to learn code embeddings [34, 31], using reinforcement learning in zero-shot setting [15, 46], bootstrapping from a small set of expert annotations [34], or using expert grammars to generate synthetic training data [52].

Neuro-symbolic program synthesis. Our approach is related to program synthesis, i.e., automatically constructing programs that satisfy a given specification [20]. The usage of deep learning models for program synthesis has resulted in significant progress in a variety of domains including string transformations [16, 14, 32], block-based visual programming [3, 4, 13, 48], and competitive programming [25]. Program synthesis has also been used to learn compositional symbolic rules and mimic abstract human learning [30, 17].

2. PROBLEM SETUP

Next, we introduce definitions and formalize our objective.

2.1 Preliminaries

The space of tasks. We define the space of tasks as $\mathbb{T}$; in this paper, $\mathbb{T}$ is inspired by the popular *Hour of Code: Maze Challenge* [9] from Code.org [8]; see Figure 1a. We define a task $\mathtt{T} \in \mathbb{T}$ as a tuple $(\mathtt{T}_{\text{vis}}, \mathtt{T}_{\text{store}}, \mathtt{T}_{\text{size}})$, where $\mathtt{T}_{\text{vis}}$ denotes a visual puzzle, $\mathtt{T}_{\text{store}}$ the available block types, and $\mathtt{T}_{\text{size}}$ the maximum number of blocks allowed in the solution

code. The task $\mathtt{T}$ in Figure 1a corresponds to Maze#18 in the *Hour of Code: Maze Challenge* [9], and has been studied in a number of prior works [35, 15, 1].

The space of codes. 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 to tasks considered in this paper, we use a DSL from [1]. A code $\mathtt{C} \in \mathbb{C}$ has the following attributes: $\mathtt{C}_{\text{blocks}}$ is the set of types of code blocks used in $\mathtt{C}$, $\mathtt{C}_{\text{size}}$ is the number of code blocks used, and $\mathtt{C}_{\text{depth}}$ is the depth of the *Abstract Syntax Tree* of $\mathtt{C}$.

Solution code and student attempt. For a given task $\mathtt{T}$, a *solution code* $\mathtt{C}^{\star}_{\mathtt{T}} \in \mathbb{C}$ should solve the visual puzzle; additionally, it can only use the allowed types of code blocks (i.e., $\mathtt{C}_{\text{blocks}} \subseteq \mathtt{T}_{\text{store}}$) and should be within the specified size threshold (i.e., $\mathtt{C}_{\text{size}} \leq \mathtt{T}_{\text{size}}$). We note that a task $\mathtt{T} \in \mathbb{T}$ may have multiple solution codes; in this paper, we typically refer to a single solution code that is provided as input. A *student attempt* for a task $\mathtt{T}$ refers to a code that is being written by a student (including incorrect or partial codes). A student attempt could be any code $\mathtt{C} \in \mathbb{C}$ as long as it uses the set of available types of code blocks (i.e., $\mathtt{C}_{\text{blocks}} \subseteq \mathtt{T}_{\text{store}}$).

2.2 Objective

Distinct phases. To formalize our objective, we introduce three distinct phases in our problem setup that provide a conceptual separation in terms of information and computation available to a system. More concretely, we have:

(1) Reference task $\mathtt{T}^{\text{ref}}$: We are given a reference task $\mathtt{T}^{\text{ref}}$ for which we have real-world datasets of different students' attempts as well as access to other data resources. Reference tasks are fixed and the system can use any computation a priori (e.g., compute code embeddings).

(2) Student $\mathtt{stu}$ attempts $\mathtt{T}^{\text{ref}}$: The system interacts with a student, namely $\mathtt{stu}$, who attempts the reference task $\mathtt{T}^{\text{ref}}$ and submits a code, denoted as $\mathtt{C}^{\mathtt{stu}}_{\mathtt{T}^{\text{ref}}}$. At the end of this phase, the system has observed $\mathtt{stu}$'s behavior on $\mathtt{T}^{\text{ref}}$ and we denote this observation by the tuple $(\mathtt{T}^{\text{ref}}, \mathtt{C}^{\mathtt{stu}}_{\mathtt{T}^{\text{ref}}})$.[1]

(3) Target task $\mathtt{T}^{\text{tar}}$: The system seeks to synthesize the student $\mathtt{stu}$'s behavior on a target task $\mathtt{T}^{\text{tar}}$. Importantly, the target task $\mathtt{T}^{\text{tar}}$ is not available a priori and this synthesis process would be done in real-time, possibly with constrained computational resources. Furthermore, the system may have to synthesize the $\mathtt{stu}$'s behavior on a large number of different target tasks from the space $\mathbb{T}$ (e.g., to personalize the next task in a curriculum).[2]

Granularity level of our objective. There are several different granularity levels at which we can predict the student $\mathtt{stu}$'s behavior for $\mathtt{T}^{\text{tar}}$, including: (a) a coarse-level binary prediction of whether $\mathtt{stu}$ will successfully solve $\mathtt{T}^{\text{tar}}$, (b) a medium-level prediction about $\mathtt{stu}$'s behavior w.r.t. to a predefined feature set (e.g., labelled misconceptions); (c) a fine-level prediction in terms of synthesizing $\mathtt{C}^{\mathtt{stu}}_{\mathtt{T}^{\text{tar}}}$, i.e., a program that $\mathtt{stu}$ would write if the system would assign $\mathtt{T}^{\text{tar}}$ to the student. In this work, we focus on this fine-level, arguably also the most challenging, synthesis objective.

Performance evaluation. So far, we have concretized the synthesis objective; however, there is still a question of how to quantitatively measure the performance of a technique set out to achieve this objective. The key challenge stems from the open-ended and conceptual nature of programming tasks. Even for seemingly simple tasks such as in Figure 1a, the students' attempts can be highly diverse, thereby making it difficult to detect a student's misconceptions from observed behaviors; moreover, the space of misconceptions itself is not clearly understood. To this end, we begin by designing a benchmark to quantitatively measure the performance of different techniques w.r.t. our objective.

3. BENCHMARK

In this section, we introduce our benchmark, StudentSyn.

3.1 StudentSyn: Data Curation

We begin by curating a synthetic dataset for the benchmark, designed to capture different scenarios of the three distinct phases mentioned in Section 2.2. In particular, each scenario corresponds to a 4-tuple $(\mathtt{T}^{\text{ref}}, \mathtt{C}^{\mathtt{stu}}_{\mathtt{T}^{\text{ref}}}, \mathtt{T}^{\text{tar}}, \mathtt{C}^{\mathtt{stu}}_{\mathtt{T}^{\text{tar}}})$, where $\mathtt{C}^{\mathtt{stu}}_{\mathtt{T}^{\text{ref}}}$ (observed by the system) and $\mathtt{C}^{\mathtt{stu}}_{\mathtt{T}^{\text{tar}}}$ (to be synthesized by the system) correspond to a student $\mathtt{stu}$'s attempts.

Reference and target tasks. We select two reference tasks for this benchmark, namely $\mathtt{T}^{4}$ and $\mathtt{T}^{18}$—they correspond to Maze#4 and Maze#18 in the *Hour of Code: Maze Challenge* [9]. These tasks have been studied in a number of prior works [35, 15, 1] because of the availability of large-scale datasets of students' attempts. For each reference task, we manually create three target tasks—Figure 2b illustrates target tasks for $\mathtt{T}^{18}$; the target tasks for $\mathtt{T}^{4}$ can be found in the longer version of the paper [47]. These target tasks are similar to the corresponding reference task in a sense that the set of available block types is same and the nesting structure of programming constructs in solution codes is same.

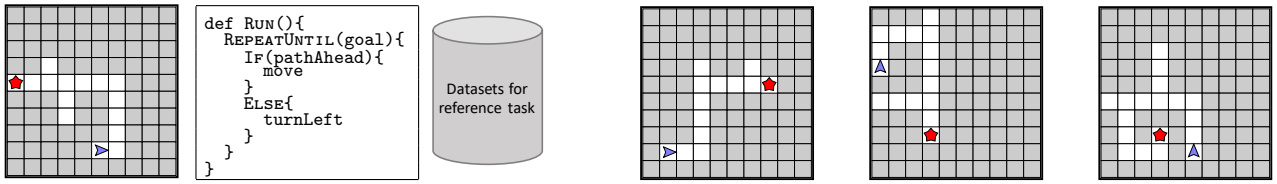
Types of students' behaviors and students' attempts. For a given reference-target task pair $(\mathtt{T}^{\text{ref}}, \mathtt{T}^{\text{tar}})$, next we seek to simulate a student $\mathtt{stu}$ to create $\mathtt{stu}$'s attempts $\mathtt{C}^{\mathtt{stu}}_{\mathtt{T}^{\text{ref}}}$ and $\mathtt{C}^{\mathtt{stu}}_{\mathtt{T}^{\text{tar}}}$. We begin by identifying a set of salient students' behaviors and misconceptions for reference tasks $\mathtt{T}^{4}$ and $\mathtt{T}^{18}$ based on students' attempts observed in the real-world dataset of [35]. In this benchmark, we select 6 types of students' behaviors for each reference task—Figure 2c highlights the 6 selected types for $\mathtt{T}^{18}$; the 6 selected types for $\mathtt{T}^{4}$ can be found in the longer version of the paper [47].[3] For a given pair $(\mathtt{T}^{\text{ref}}, \mathtt{T}^{\text{tar}})$, we first simulate a student $\mathtt{stu}$ by associating this student to one of the 6 types, and then manually create $\mathtt{stu}$'s attempts $\mathtt{C}^{\mathtt{stu}}_{\mathtt{T}^{\text{ref}}}$ and $\mathtt{C}^{\mathtt{stu}}_{\mathtt{T}^{\text{tar}}}$. For a given scenario $(\mathtt{T}^{\text{ref}}, \mathtt{C}^{\mathtt{stu}}_{\mathtt{T}^{\text{ref}}}, \mathtt{T}^{\text{tar}}, \mathtt{C}^{\mathtt{stu}}_{\mathtt{T}^{\text{tar}}})$, the attempt $\mathtt{C}^{\mathtt{stu}}_{\mathtt{T}^{\text{tar}}}$ is not observed and serves as a *ground truth* for evaluation purposes; henceforth, we interchangeably write a scenario as $(\mathtt{T}^{\text{ref}}, \mathtt{C}^{\mathtt{stu}}_{\mathtt{T}^{\text{ref}}}, \mathtt{T}^{\text{tar}}, ?)$.

Total scenarios. We create 72 scenarios $(\mathtt{T}^{\text{ref}}, \mathtt{C}^{\mathtt{stu}}_{\mathtt{T}^{\text{ref}}}, \mathtt{T}^{\text{tar}}, \mathtt{C}^{\mathtt{stu}}_{\mathtt{T}^{\text{tar}}})$ in the benchmark corresponding to (i) 2 reference tasks, (ii) 3 target tasks per reference task, (iii) 6 types of students' behaviors per reference task, and (iv) 2 students per type.

[1] In practice, the system might have more information, e.g., the whole trajectory of edits leading to $\mathtt{C}^{\mathtt{stu}}_{\mathtt{T}^{\text{ref}}}$.

[2] Even though the *Hour of Code: Maze Challenge* [9] has only 20 tasks, the space $\mathbb{T}$ is intractably large and new tasks can be generated, e.g., for providing feedback [1].

[3] We note that, in real-world settings, the types of students' behaviors and their attempts have a much larger variability and complexities with a long-tail distribution.



(a) Reference task $\mathtt{T}^{18}$ with solution code and datasets

(b) Three target tasks for $\mathtt{T}^{18}$: $\mathtt{T}^{18x}$, $\mathtt{T}^{18y}$, and $\mathtt{T}^{18z}$

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

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

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

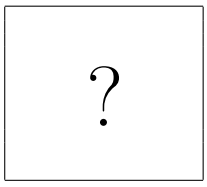
```
def Run(){
  RepeatUntil(goal){
    move
    turnLeft
    move
    turnLeft
    move
  }
}
```

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

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

(c) Example codes (i)–(vi) corresponding to six types of students' behaviors when attempting $\mathtt{T}^{18}$, each capturing different misconceptions

Figure 2: Illustration of the key elements of the StudentSyn benchmark for the reference task $\mathtt{T}^{18}$ shown in **(a)**—same as in Figure 1a. **(b)** Shows three target tasks associated with $\mathtt{T}^{18}$; these target tasks are similar to $\mathtt{T}^{18}$ in a sense that the set of available block types is same as $\mathtt{T}^{18}_{\text{store}}$ and the nesting structure of programming constructs in solution codes is same as in $\mathtt{C}^{*}_{\mathtt{T}^{18}}$. **(c)** Shows example codes corresponding to six types of students' behaviors when attempting $\mathtt{T}^{18}$, each capturing a different misconception as follows: (i) confusing left/right directions when turning or checking conditionals, (ii) following one of the wrong path segments, (iii) misunderstanding of `IfElse` structure functionality and writing the same blocks in both the execution branches, (iv) ignoring the `IfElse` structure when solving the task, (v) ignoring the `While` structure when solving the task, (vi) attempting to solve the task by using only the basic action blocks in {`turnLeft`, `turnRight`, `move`}.



`stu`'s attempt for $\mathtt{T}^{18x}$ in Figure 1

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

option (a)

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

option (b)

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

option (c)

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

option (d)

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

option (e)

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

option (f)

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

option (g)

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

option (h)

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

option (i)

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

option (j)

Figure 3: Illustration of the generative and discriminative objectives in the StudentSyn benchmark for the scenario shown in Figure 1. For the generative objective, the goal is to synthesize the student `stu`'s behavior on the target task $\mathtt{T}^{18x}$, i.e., a program that `stu` would write if the system would assign $\mathtt{T}^{18x}$ to the student. For the discriminative objective, the goal is to choose one of the ten codes, shown as options (a)–(j), that corresponds to the student `stu`'s attempt. For each scenario, ten options are created systematically as discussed in Section 3.2; in this illustration, option (a) corresponds to the solution code $\mathtt{C}^{*}_{\mathtt{T}^{18x}}$ for the target task and option (e) corresponds to the student `stu`'s attempt as designed in the benchmark.

3.2 StudentSyn: Performance Measures

We introduce two performance measures to capture our synthesis objective. Our first measure, namely *generative performance*, is to directly capture the quality of fine-level synthesis of the student `stu`'s attempt—this measure requires a human-in-the-loop evaluation. To further automate the evaluation process, we then introduce a second performance measure, namely *discriminative performance*.

Generative performance. As a generative performance measure, we introduce a 4-point *Likert scale* to evaluate the quality of synthesizing `stu`'s attempt $\mathtt{C}^{\mathtt{stu}}_{\mathtt{T}^{\text{tar}}}$ for a scenario $(\mathtt{T}^{\text{ref}}, \mathtt{C}^{\mathtt{stu}}_{\mathtt{T}^{\text{ref}}}, \mathtt{T}^{\text{tar}}, ?)$. The scale is designed to assign scores based on two factors: (a) whether the elements of the student's behavior observed in $\mathtt{C}^{\mathtt{stu}}_{\mathtt{T}^{\text{ref}}}$ are present, (b) whether the elements of the target task $\mathtt{T}^{\text{tar}}$ (e.g., parts of its solution) are present. More concretely, the scores are assigned as

follows (with higher scores being better): (i) Score 1 means the technique does not have synthesis capability; (ii) Score 2 means the synthesis fails to capture the elements of $\mathtt{C}^{\mathtt{stu}}_{\mathtt{T}^{\mathrm{ref}}}$ and $\mathtt{T}^{\mathrm{tar}}$; (iii) Score 3 means the synthesis captures the elements only of $\mathtt{C}^{\mathtt{stu}}_{\mathtt{T}^{\mathrm{ref}}}$ or of $\mathtt{T}^{\mathrm{tar}}$, but not both; (iv) Score 4 means the synthesis captures the elements of both $\mathtt{C}^{\mathtt{stu}}_{\mathtt{T}^{\mathrm{ref}}}$ and $\mathtt{T}^{\mathrm{tar}}$.

Discriminative performance. As the generative performance measure requires human-in-the-loop evaluation, we also introduce a disciminative performance measure based on the prediction accuracy of choosing the student attempt from a set. More concretely, given a scenario $(\mathtt{T}^{\mathrm{ref}}, \mathtt{C}^{\mathtt{stu}}_{\mathtt{T}^{\mathrm{ref}}}, \mathtt{T}^{\mathrm{tar}}, ?)$, the discriminative objective is to choose $\mathtt{C}^{\mathtt{stu}}_{\mathtt{T}^{\mathrm{tar}}}$ from ten candidate codes; see Figure 3. These ten options are created automatically in a systematic way and include: (a) the *ground-truth* $\mathtt{C}^{\mathtt{stu}}_{\mathtt{T}^{\mathrm{tar}}}$, (b) the solution code $\mathtt{C}^{\star}_{\mathtt{T}^{\mathrm{tar}}}$, (c) five codes $\mathtt{C}^{\mathtt{stu}'}_{\mathtt{T}^{\mathrm{tar}}}$ from the benchmark associated with other students $\mathtt{stu}'$ whose behavior type is different from $\mathtt{stu}$, and (iv) three *randomly* constructed codes obtained by editing $\mathtt{C}^{*}_{\mathtt{T}^{\mathrm{tar}}}$.

4. METHODOLOGY

In this section, we design different techniques for the benchmark StudentSyn. First, we consider a few simple baselines for the discriminative-only objective (RandD, EditD, EditEmbD). Next, we develop our two main techniques inspired by neural/symbolic methods (NeurSS, SymSS). Finally, we propose performance evaluation of human experts (TutorSS). Table 1 illustrates how these techniques differ in required inputs and domain knowledge. Below, we provide a brief overview of these techniques; we refer the reader to the longer version of the paper for full details [47].

Simple baselines. As a starting point, we consider simple baselines for the discriminative-only objective; they do not have synthesis capability. Our first baseline RandD simply chooses a code from the 10 options at random. Our next two baselines, EditD and EditEmbD, are defined through a distance function $\mathrm{D}_{\mathtt{T}^{\mathrm{ref}}}(\mathtt{C}, \mathtt{C}')$ that quantifies a notion of distance between any two codes $\mathtt{C}, \mathtt{C}'$ for a fixed reference task. For a scenario $(\mathtt{T}^{\mathrm{ref}}, \mathtt{C}^{\mathtt{stu}}_{\mathtt{T}^{\mathrm{ref}}}, \mathtt{T}^{\mathrm{tar}}, ?)$ and ten option codes, these baselines select the code $\mathtt{C}$ that minimizes $\mathrm{D}_{\mathtt{T}^{\mathrm{ref}}}(\mathtt{C}, \mathtt{C}^{\mathtt{stu}}_{\mathtt{T}^{\mathrm{ref}}})$. EditD uses a tree-edit distance between *Abstract Syntax Trees* as the distance function, denoted as $\mathrm{D}^{\mathrm{edit}}_{\mathtt{T}^{\mathrm{ref}}}$. EditEmbD extends EditD by considering a distance function that combines $\mathrm{D}^{\mathrm{edit}}_{\mathtt{T}^{\mathrm{ref}}}$ and a code-embedding based distance function $\mathrm{D}^{\mathrm{emb}}_{\mathtt{T}^{\mathrm{ref}}}$; in this paper, we trained code embeddings with the methodology of [15] using a real-world dataset of student attempts on $\mathtt{T}^{\mathrm{ref}}$. EditEmbD then uses a distance function as a convex combination $\big(\alpha \cdot \mathrm{D}^{\mathrm{edit}}_{\mathtt{T}^{\mathrm{ref}}}(\mathtt{C}, \mathtt{C}') + (1-\alpha) \cdot \mathrm{D}^{\mathrm{emb}}_{\mathtt{T}^{\mathrm{ref}}}(\mathtt{C}, \mathtt{C}')\big)$ where α is optimized for each reference task separately.

Neural synthesizer NeurSS. Next, we develop our technique, NeurSS (*Neural Program Synthesis for* StudentSyn), inspired by recent advances in neural program synthesis [3, 4]. A neural synthesizer model takes as input a visual task $\mathtt{T}$, and then sequentially synthesizes a code $\mathtt{C}$ by using programming tokens in $\mathtt{T}_{\mathrm{store}}$. However, our goal is not simply to synthesize a solution code for the input task $\mathtt{T}$ as considered in [3, 4]; instead, we want to synthesize attempts of a given student that the system is interacting with at real-time/deployment. To achieve this goal, NeurSS operates in three stages, where each stage is in line with a phase of our objective described in Section 2.2. At a high-level, the three stages of NeurSS are as follows: (i) In Stage-1, we are given a reference task and its solution $(\mathtt{T}^{\mathrm{ref}}, \mathtt{C}^{\star}_{\mathtt{T}^{\mathrm{ref}}})$, and train a neural synthesizer model that can synthesize solutions for any task similar to $\mathtt{T}^{\mathrm{ref}}$; (ii) In Stage-2, the system observes the student $\mathtt{stu}$'s attempt $\mathtt{C}^{\mathtt{stu}}_{\mathtt{T}^{\mathrm{ref}}}$ and initiates *continual training* of the neural synthesizer model from Stage-1 in real-time; (iii) In Stage-3, the system considers a target task $\mathtt{T}^{\mathrm{tar}}$ and uses the model from Stage-2 to synthesize $\mathtt{C}^{\mathtt{stu}}_{\mathtt{T}^{\mathrm{tar}}}$.

Symbolic synthesizer SymSS. As we will see in experiments, NeurSS significantly outperforms the simple baselines introduced earlier; yet, there is a substantial gap in the performance of NeurSS and human experts (i.e., TutorSS). An important question that we seek to resolve is how much of this performance gap can be reduced by leveraging domain knowledge such as how students with different behaviors (misconceptions) write codes. To this end, we develop our technique, SymSS (*Symbolic Program Synthesis for* StudentSyn), inspired by recent advances in using symbolic methods for program synthesis [24, 52, 1, 26]. Similar in spirit to NeurSS, SymSS operates in three stages as follows: (i) In Stage-1, we are given $(\mathtt{T}^{\mathrm{ref}}, \mathtt{C}^{\star}_{\mathtt{T}^{\mathrm{ref}}})$, and design a symbolic synthesizer model using *Probabilistic Context Free Grammars* (PCFG) to encode how students of different behavior types $\mathcal{M}$ write codes for any task similar to $\mathtt{T}^{\mathrm{ref}}$ [5, 27, 52]; (ii) In Stage-2, the system observes the student $\mathtt{stu}$'s attempt $\mathtt{C}^{\mathtt{stu}}_{\mathtt{T}^{\mathrm{ref}}}$ and makes a prediction about the behavior type $\mathrm{M}^{\mathtt{stu}} \in \mathcal{M}$; (iii) In Stage-3, the system considers a target task $\mathtt{T}^{\mathrm{tar}}$ and uses the model from Stage-1 to synthesize $\mathtt{C}^{\mathtt{stu}}_{\mathtt{T}^{\mathrm{tar}}}$ based on the inferred $\mathrm{M}^{\mathtt{stu}}$.

Human experts. Finally, we propose an evaluation of human experts' performance on the benchmark StudentSyn, and refer to this evaluation technique as TutorSS. These evaluations are done through a web platform where an expert would provide a generative or discriminative response to a given scenario $(\mathtt{T}^{\mathrm{ref}}, \mathtt{C}^{\mathtt{stu}}_{\mathtt{T}^{\mathrm{ref}}}, \mathtt{T}^{\mathrm{tar}}, ?)$. In our work, TutorSS involved participation of three independent experts for the evaluation—these experts have had experience in block-based programming and tutoring. We first carry out generative evaluations where an expert has to write the student attempt code; afterwards, we carry out discriminative evaluations where an expert would choose one of the options.

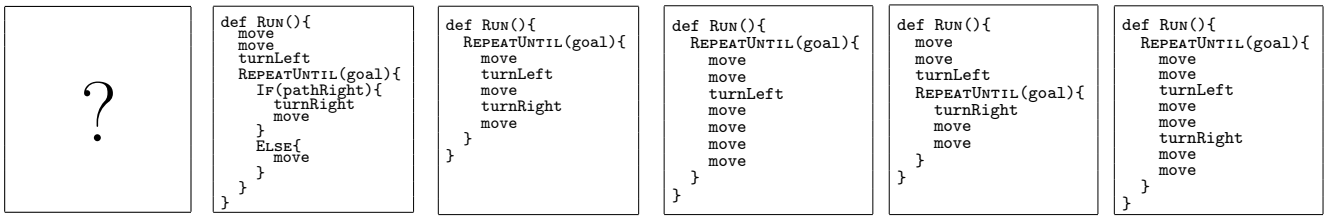
5. EXPERIMENTAL EVALUATION

In this section, we evaluate the performance of different techniques discussed in Section 4. Our results are summarized in Table 1 and Figure 4. Below, we provide a brief overview of the evaluation procedures and results; we refer the reader to the longer version of the paper for full details [47].

Generative performance. As discussed in Section 3.2, we evaluate the generative performance of a technique in the following steps: (a) a scenario $(\mathtt{T}^{\mathrm{ref}}, \mathtt{C}^{\mathtt{stu}}_{\mathtt{T}^{\mathrm{ref}}}, \mathtt{T}^{\mathrm{tar}}, ?)$ is picked; (b) the technique synthesizes $\mathtt{stu}$'s attempt; (c) the generated code is scored on the 4-point *Likert scale*. The scoring step requires human-in-the-loop evaluation and involved an expert (different from the three experts that are part of TutorSS). Overall, each technique is evaluated for 36 unique scenarios in StudentSyn—we selected 18 scenarios per reference task by first picking one of the 3 target tasks and then picking a student from one of the 6 different types of behavior. The final performance results in Table 1 are re-

Method	Generative Performance		Discriminative Performance		Required Inputs and Domain Knowledge				
	Reference task $\mathtt{T}^4$	Reference task $\mathtt{T}^{18}$	Reference task $\mathtt{T}^4$	Reference task $\mathtt{T}^{18}$	Ref. task dataset: student attempts	Ref. task dataset: similar tasks	Student types	Expert grammars	Expert evaluation
RandD	1.00	1.00	10.0	10.0	-	-	-	-	-
EditD	1.00	1.00	31.5	48.9	-	-	-	-	-
EditEmbD	1.00	1.00	39.6	48.9	✗	-	-	-	-
NeurSS	3.00	2.83	43.8	57.2	✗	✗	-	-	-
SymSS	3.78	3.72	88.1	62.1	-	-	✗	✗	-
TutorSS	3.85	3.90	89.8	85.2	-	-	-	-	✗

Table 1: This table shows results on StudentSyn in terms of the generative and discriminative performance measures. The columns under "Required Inputs and Domain Knowledge" highlight information used by different techniques (✗ indicates the usage of the corresponding input/knowledge). The values are in the range $[1.0, 4.0]$ for generative performance and in the range $[0.0, 100.0]$ for discriminative performance—higher values being better. Human experts (TutorSS) can achieve high performance on both the measures, whereas simple baselines perform poorly. NeurSS and SymSS significantly improve upon the simple baselines; yet, there is a high gap in performance in comparison to that of human experts.



(a) Attempt $\mathtt{C}^{\mathtt{stu}}_{\mathtt{T}^{18x}}$ (b) Solution $\mathtt{C}^{\star}_{\mathtt{T}^{18x}}$ (c) Benchmark code (d) NeurSS (e) SymSS (f) TutorSS

Figure 4: Qualitative results for the scenario in Figure 1. **(a)** The goal is to synthesize the student `stu`'s behavior on the target task $\mathtt{T}^{18x}$. **(b)** Solution code $\mathtt{C}^{\star}_{\mathtt{T}^{18x}}$ for the target task. **(c)** Code provided in the benchmark as a possible answer for this scenario. **(d, e)** Codes synthesized by our techniques NeurSS and SymSS. **(f)** Code provided by one of the human experts.

ported as an average across these scenarios; for TutorSS, each of the three experts independently responded to these 36 scenarios and the final performance is averaged across experts. The simple baselines (RandD, EditD, EditEmbD) have a score of 1.00 as they do not have a synthesis capability. TutorSS achieves the highest performance; SymSS also achieves high performance (only slightly lower than that of TutorSS)—the high performance of SymSS is expected given its knowledge about types of students in StudentSyn and the expert domain knowledge inherent in its design. NeurSS improves upon simple baselines, but performs worse compared to SymSS and TutorSS. Figure 4 illustrates the codes generated by different techniques for the scenario in Figure 1—the codes by TutorSS and SymSS are high-scoring w.r.t. our 4-point *Likert scale*; however, the code by NeurSS only captures elements of the student's behavior in $\mathtt{C}^{\mathtt{stu}}_{\mathtt{T}^{\mathrm{ref}}}$ but misses elements of the target task $\mathtt{T}^{\mathrm{tar}}$. We provide additional details and statistical significance results w.r.t. χ^2 test [6] in the longer version of the paper [47].

Discriminative performance. As discussed in Section 3.2, we evaluate the discriminative performance of a technique in the following steps: (a) a discriminative instance is created with a scenario $(\mathtt{T}^{\mathrm{ref}}, \mathtt{C}^{\mathtt{stu}}_{\mathtt{T}^{\mathrm{ref}}}, \mathtt{T}^{\mathrm{tar}}, ?)$ picked from the benchmark and 10 code options created automatically; (b) the technique chooses one of the options as `stu`'s attempt; (c) the chosen option is scored either 100.0 when correct, or 0.0 otherwise. For all techniques except TutorSS, we perform evaluation on a set of 720 instances (360 instances per reference task); for TutorSS, we perform evaluation on a small set of 72 instances (36 instances per reference task), to reduce the effort for human experts. The final performance results in Table 1 are reported as an average predictive accuracy across the evaluated instances; for TutorSS, each of the three experts independently responded to the instances and the final performance is averaged across experts. Results highlight the huge performance gap between the human experts (TutorSS) and simple baselines (RandD, EditD, EditEmbD). Our proposed techniques (NeurSS and SymSS) have substantially reduced this performance gap w.r.t. TutorSS. SymSS achieves high performance compared to simple baselines and NeurSS; moreoever, on the reference task $\mathtt{T}^4$, its performance is close to that of TutorSS. The high performance of SymSS is partly due to its access to types of students in StudentSyn; in fact, this information is used only by SymSS and is not even available to human experts in TutorSS (see column "Student types" in Table 1). NeurSS outperformed simple baselines but its performance is below SymSS and TutorSS. We provide additional details and statistical significance results w.r.t. Tukey's HSD test [49] in the longer version of the paper [47].

6. CONCLUSIONS

We investigated student modeling in the context of block-based visual programming environments, focusing on the ability to automatically infer students' misconceptions and synthesize their expected behavior. We introduced a novel benchmark, StudentSyn, to objectively measure the generative as well as the discriminative performance of different techniques. The gap in performance between human experts (TutorSS) and our techniques (NeurSS, SymSS) highlights the challenges in synthesizing student attempts for programming tasks. We believe that the benchmark will facilitate further research in this crucial area of student modeling for block-based visual programming environments.

7. ACKNOWLEDGMENTS

This work was supported in part by the European Research Council (ERC) under the Horizon Europe programme (ERC StG, grant agreement No. 101039090).

References

[1] U. Z. Ahmed, M. Christakis, A. Efremov, N. Fernandez, A. Ghosh, A. Roychoudhury, and A. Singla. Synthesizing Tasks for Block-based Programming. In *NeurIPS*, 2020.

[2] F. Ai, Y. Chen, Y. Guo, Y. Zhao, Z. Wang, G. Fu, and G. Wang. Concept-Aware Deep Knowledge Tracing and Exercise Recommendation in an Online Learning System. In *EDM*, 2019.

[3] R. Bunel, M. J. Hausknecht, J. Devlin, R. Singh, and P. Kohli. Leveraging Grammar and Reinforcement Learning for Neural Program Synthesis. In *ICLR*, 2018.

[4] X. Chen, C. Liu, and D. Song. Execution-Guided Neural Program Synthesis. In *ICLR*, 2019.

[5] N. Chomsky. On Certain Formal Properties of Grammars. *Information and control*, 2:137–167, 1959.

[6] W. G. Cochran. The $\chi 2$ Test of Goodness of Fit. *The Annals of Mathematical Statistics*, pages 315–345, 1952.

[7] J. Cock, M. Marras, C. Giang, and T. Käser. Early Prediction of Conceptual Understanding in Interactive Simulations. In *EDM*, 2021.

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

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

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

[11] A. T. Corbett and J. R. Anderson. Knowledge Tracing: Modeling the Acquisition of Procedural Knowledge. *User Modeling and User-Adapted Interaction*, 4(4):253–278, 1994.

[12] A. T. Corbett, M. McLaughlin, and K. C. Scarpinatto. Modeling Student Knowledge: Cognitive Tutors in High School and College. *User Model. User Adapt. Interact.*, 2000.

[13] J. Devlin, R. Bunel, R. Singh, M. J. Hausknecht, and P. Kohli. Neural Program Meta-Induction. In *NeurIPS*, 2017.

[14] J. Devlin, J. Uesato, S. Bhupatiraju, R. Singh, A. Mohamed, and P. Kohli. Robustfill: Neural Program Learning under Noisy I/O. In D. Precup and Y. W. Teh, editors, *ICML*, 2017.

[15] A. Efremov, A. Ghosh, and A. Singla. Zero-shot Learning of Hint Policy via Reinforcement Learning and Program Synthesis. In *EDM*, 2020.

[16] K. Ellis, M. I. Nye, Y. Pu, F. Sosa, J. Tenenbaum, and A. Solar-Lezama. Write, Execute, Assess: Program Synthesis with a REPL. In *NeurIPS*, 2019.

[17] K. Ellis, C. Wong, M. I. Nye, M. Sablé-Meyer, L. Cary, L. Morales, L. B. Hewitt, A. Solar-Lezama, and J. B. Tenenbaum. Dreamcoder: Growing Generalizable, Interpretable Knowledge with Wake-Sleep Bayesian Program Learning. *CoRR*, abs/2006.08381, 2020.

[18] A. Emerson, A. Smith, F. J. Rodríguez, E. N. Wiebe, B. W. Mott, K. E. Boyer, and J. C. Lester. Cluster-Based Analysis of Novice Coding Misconceptions in Block-Based Programming. In *SIGCSE*, 2020.

[19] A. Ghosh, S. Tschiatschek, S. Devlin, and A. Singla. Adaptive Scaffolding in Block-based Programming via Synthesizing New Tasks as Pop Quizzes. In *AIED*, 2022.

[20] S. Gulwani, O. Polozov, and R. Singh. Program Synthesis. *Foundations and Trends® in Programming Languages*, 2017.

[21] J. He-Yueya and A. Singla. Quizzing Policy Using Reinforcement Learning for Inferring the Student Knowledge State. In *EDM*, 2021.

[22] A. Hunziker, Y. Chen, O. M. Aodha, M. G. Rodriguez, A. Krause, P. Perona, Y. Yue, and A. Singla. Teaching Multiple Concepts to a Forgetful Learner. In *NeurIPS*, 2019.

[23] T. Käser and D. L. Schwartz. Modeling and Analyzing Inquiry Strategies in Open-Ended Learning Environments. *Journal of AIED*, 30(3):504–535, 2020.

[24] B. M. Lake, R. Salakhutdinov, and J. B. Tenenbaum. Human-level Concept Learning through Probabilistic Program Induction. *Science*, 2015.

[25] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, J. Keeling, F. Gimeno, A. D. Lago, T. Hubert, P. Choy, and C. de. Competition-Level Code Generation with AlphaCode. 2022.

[26] A. Malik, M. Wu, V. Vasavada, J. Song, M. Coots, J. Mitchell, N. D. Goodman, and C. Piech. Generative Grading: Near Human-level Accuracy for Automated Feedback on Richly Structured Problems. In *EDM*, 2021.

[27] J. C. Martin. *Introduction to Languages and the Theory of Computation*, volume 4. McGraw-Hill NY, 1991.

[28] R. McIlroy-Young, S. Sen, J. M. Kleinberg, and A. Anderson. Aligning Superhuman AI with Human Behavior: Chess as a Model System. In *KDD*, 2020.

[29] R. McIlroy-Young and R. Wang. Detecting Individual Decision-Making Style: Exploring Behavioral Stylometry in Chess. In *NeurIPS*, 2021.

[30] M. I. Nye, A. Solar-Lezama, J. Tenenbaum, and B. M. Lake. Learning Compositional Rules via Neural Program Synthesis. In *NeurIPS*, 2020.

[31] B. Paaßen, B. Hammer, T. W. Price, T. Barnes, S. Gross, and N. Pinkwart. The Continuous Hint Factory - Providing Hints in Continuous and Infinite Spaces. *Journal of Educational Data Mining*, 2018.

[32] E. Parisotto, A. Mohamed, R. Singh, L. Li, D. Zhou, and P. Kohli. Neuro-Symbolic Program Synthesis. In *ICLR*, 2017.

[33] C. Piech, J. Bassen, J. Huang, S. Ganguli, M. Sahami, L. J. Guibas, and J. Sohl-Dickstein. Deep Knowledge Tracing. In *NeurIPS*, pages 505–513, 2015.

[34] C. Piech, J. Huang, A. Nguyen, M. Phulsuksombati, M. Sahami, and L. J. Guibas. Learning Program Embeddings to Propagate Feedback on Student Code. In *ICML*, 2015.

[35] C. Piech, M. Sahami, J. Huang, and L. J. Guibas. Autonomously Generating Hints by Inferring Problem Solving Policies. In *L@S*, 2015.

[36] L. Portnoff, E. N. Gustafson, K. Bicknell, and J. Rollinson. Methods for Language Learning Assessment at Scale: Duolingo Case Study. In *EDM*, 2021.

[37] T. W. Price and T. Barnes. Position paper: Block-based Programming Should Offer Intelligent Support for Learners. In *2017 IEEE Blocks and Beyond Workshop (B B)*, 2017.

[38] T. W. Price, Y. Dong, and D. Lipovac. iSnap: Towards Intelligent Tutoring in Novice Programming Environments. In *SIGCSE*, pages 483–488, 2017.

[39] T. W. Price, R. Zhi, and T. Barnes. Evaluation of a Data-driven Feedback Algorithm for Open-ended Programming. *EDM*, 2017.

[40] A. N. Rafferty, R. Jansen, and T. L. Griffiths. Using Inverse Planning for Personalized Feedback. In *EDM*, 2016.

[41] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, et al. Scratch: Programming for All. *Communications of the ACM*, 2009.

[42] B. Settles and B. Meeder. A Trainable Spaced Repetition Model for Language Learning. In *ACL*, 2016.

[43] A. Shakya, V. Rus, and D. Venugopal. Student Strategy Prediction using a Neuro-Symbolic Approach. *EDM*, 2021.

[44] Y. Shi, K. Shah, W. Wang, S. Marwan, P. Penmetsa, and T. W. Price. Toward Semi-Automatic Misconception Discovery Using Code Embeddings. In *LAK*, 2021.

[45] R. Singh, S. Gulwani, and A. Solar-Lezama. Automated Feedback Generation for Introductory Programming Assignments. In *PLDI*, pages 15–26, 2013.

[46] A. Singla, A. N. Rafferty, G. Radanovic, and N. T. Heffernan. Reinforcement Learning for Education: Opportunities and Challenges. *CoRR*, abs/2107.08828, 2021.

[47] A. Singla and N. Theodoropoulos. From {Solution Synthesis} to {Student Attempt Synthesis} for Block-Based Visual Programming Tasks. *CoRR*, abs/2205.01265, 2022.

[48] D. Trivedi, J. Zhang, S. Sun, and J. J. Lim. Learning to Synthesize Programs as Interpretable and Generalizable policies. *CoRR*, abs/2108.13643, 2021.

[49] J. W. Tukey. Comparing Individual Means in the Analysis of Variance. *Biometrics*, 5 2:99–114, 1949.

[50] L. Wang, A. Sy, L. Liu, and C. Piech. Learning to Represent Student Knowledge on Programming Exercises Using Deep Learning. In *EDM*, 2017.

[51] D. Weintrop and U. Wilensky. Comparing Block-based and Text-based Programming in High School Computer Science Classrooms. *ACM Transactions of Computing Education*, 18(1):1–25, 2017.

[52] M. Wu, M. Mosse, N. D. Goodman, and C. Piech. Zero Shot Learning for Code Education: Rubric Sampling with Deep Learning Inference. In *AAAI*, 2019.

[53] J. Yi, U. Z. Ahmed, A. Karkare, S. H. Tan, and A. Roychoudhury. A Feasibility Study of Using Automated Program Repair for Introductory Programming Assignments. In *ESEC/FSE*, 2017.