A K-8 Debugging Learning Trajectory Derived from Research Literature

Paper presented at the
ACM SIGCSE Technical Symposium

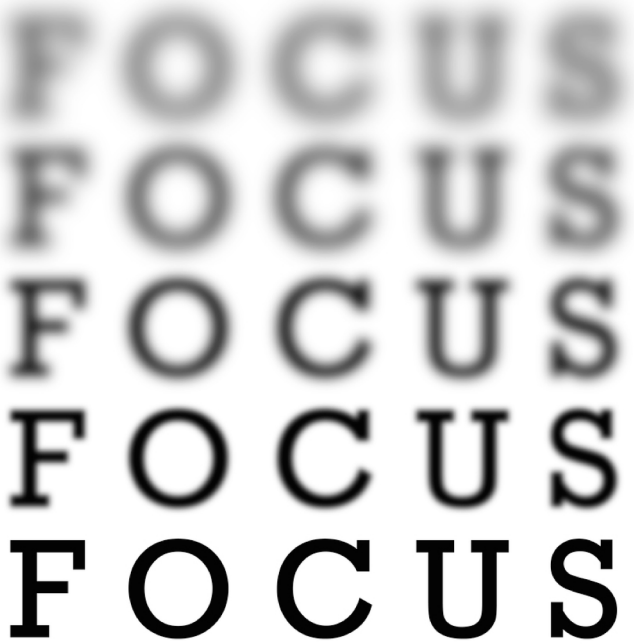
Friday, March 1, 2019

Katie Rich
Andrew Binkowski
Carla Strickland
Diana Franklin

MICHIGAN STATE UNIVERSITY
UCHICAGO STEM EDUCATION

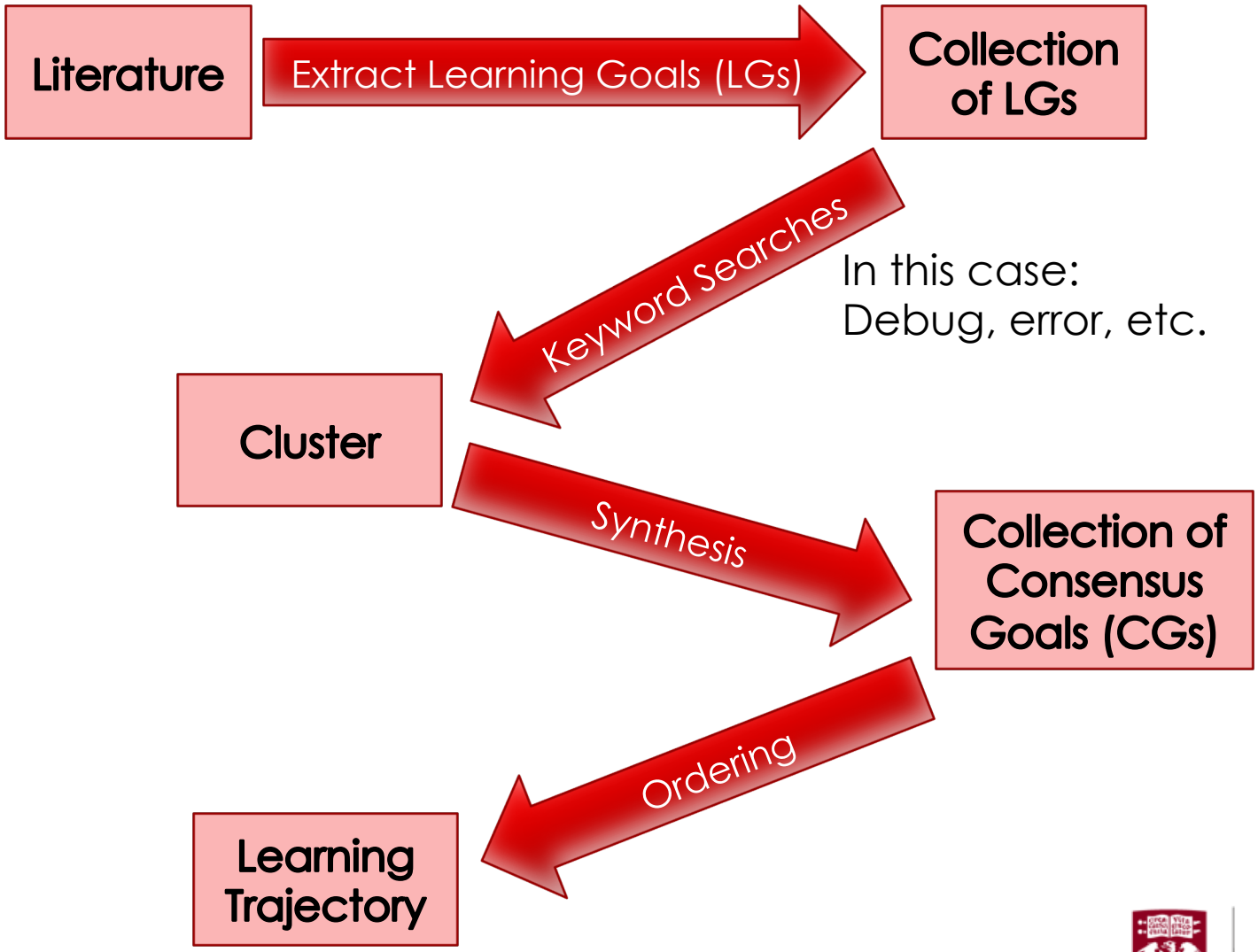
Context: The LTEC Project

- LTEC = Learning Trajectories for Everyday Computing
- What big ideas in computer science should be addressed in K-5 curricula?
- How can we express them in ways that make sense to K-5 students and teachers?
- How should they grow across time?



UCHICAGO STEM EDUCATION

Overview of Methods



UCHICAGO STEM EDUCATION

In other papers…

- We shared an overall picture of our lit review:
  - SIGCSE 2017: Learning Goals, Theorized vs. Researched
- We focused on the details of our LT process….
  - ICER 2017: Sequence, Repetition, Conditionals
- We unpacked a specific LT and its construction…
  - ICER 2018: Decomposition
- This paper & presentation focuses on:
  - Articulating the unarticulated
  - Early use of the LT for illustrative activity development
  - Dimensions of practice as an organizing structure

UCHICAGO STEM EDUCATION

Learning Goals:

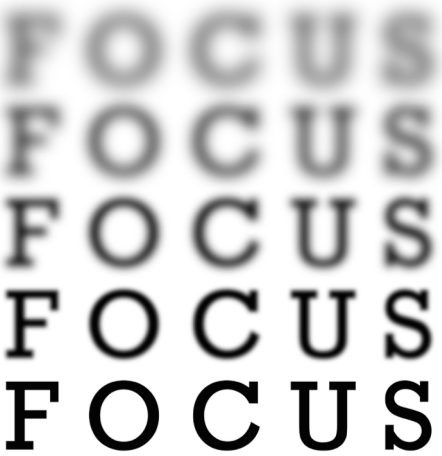
"Determine that the algorithm produces the right answer (correctness)."

"Recognize when instructions do not correspond to actions."

"determine if an artificial entity is behaving rationally"

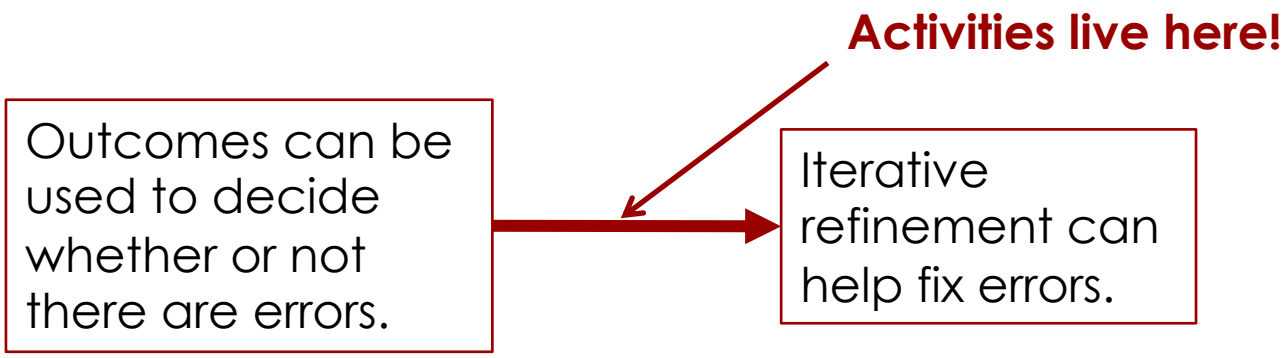
Recognize "a problem with a current solution".

Consensus Goal:
Outcomes can be used to decide whether or not there are errors.



Where to next?

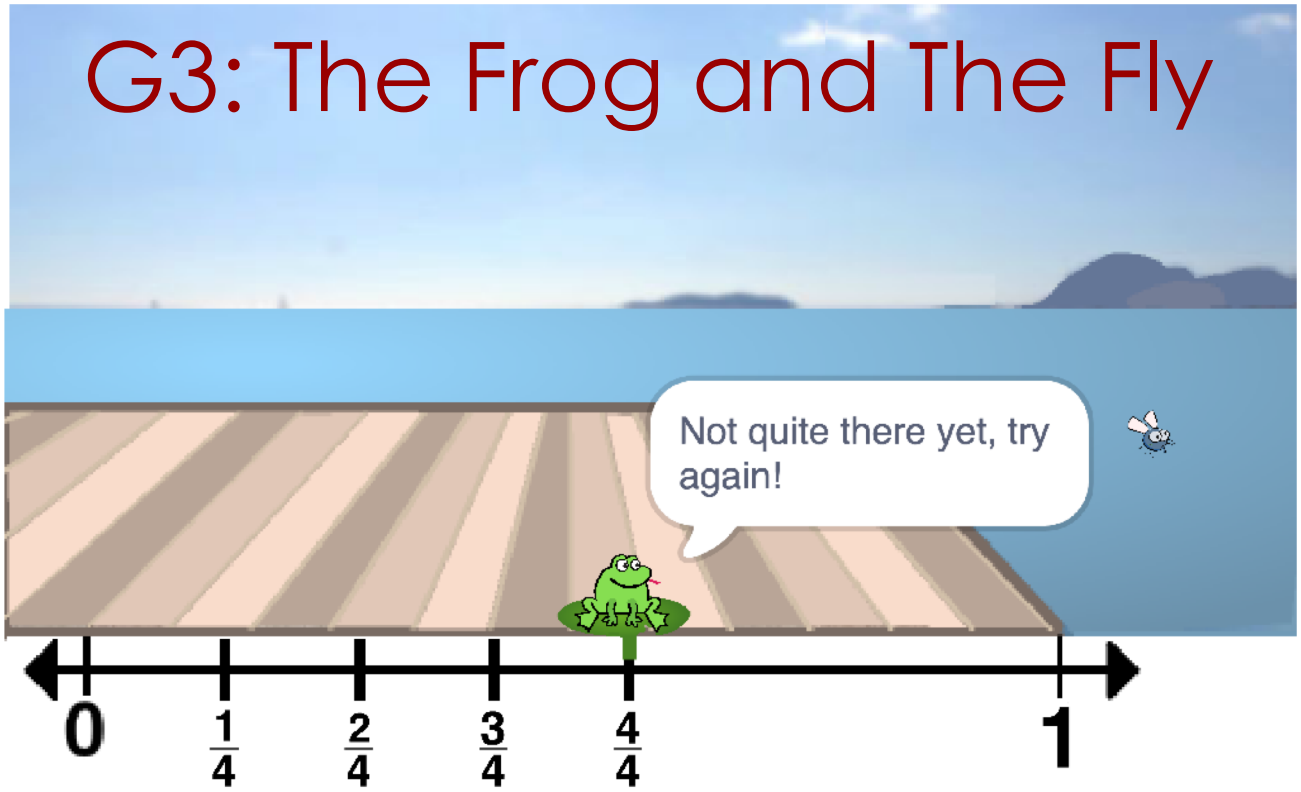
- We can embed discussion of that CG in mathematics instruction.
- What could be addressed first in computing instruction?
- The lit said:
  - PK-K students engaged in trial-and-error refinement (if they realized there was a problem!)
    (Fessakis, Gouli, &Mavroudi, 2013; Flannery& Bers, 2013)



What does this look like for elementary students?

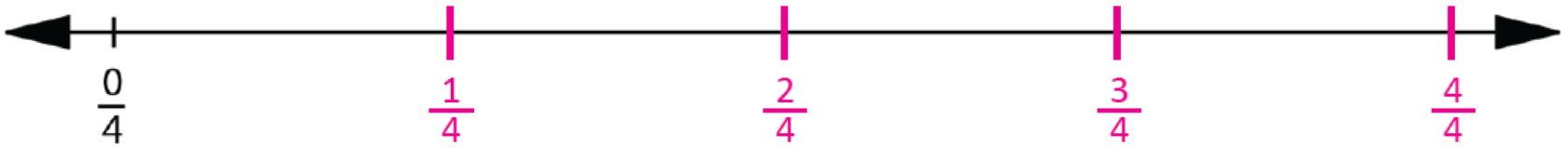
Outcomes can be used to decide whether or not there are errors.

Iterative refinement can help fix errors.

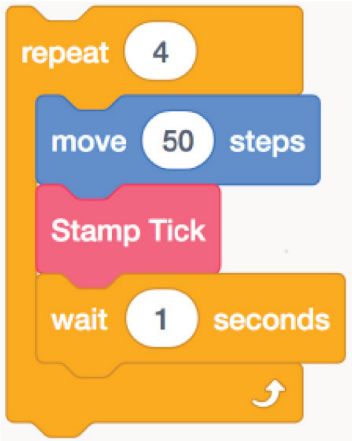


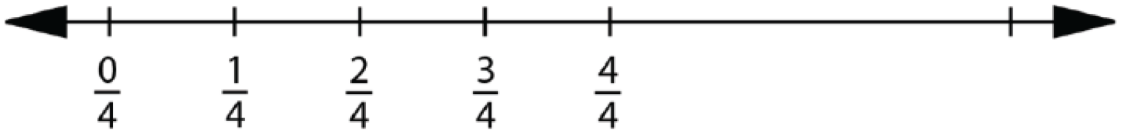
Debugging Activity: Student Page (excerpt)

① Draw tick marks to partition the line segment below into fourths. Label each fraction tick mark.



② We want the program to partition the line segment into fourths. The line segment below is 360 steps long. When the code below runs, it gives this output:





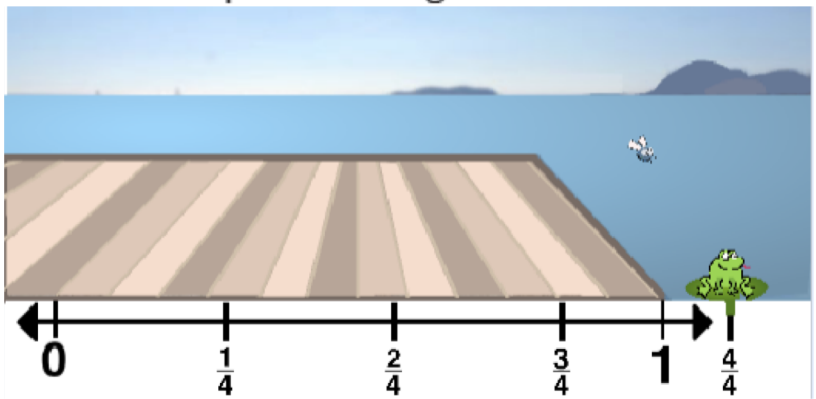
Are the spaces between the tick marks too big or too small?

too big **too small**

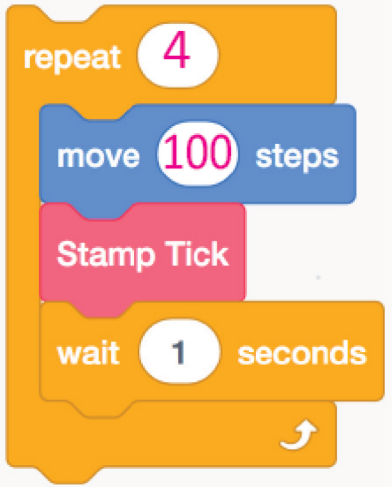
Note: Sample student work shown in **magenta**

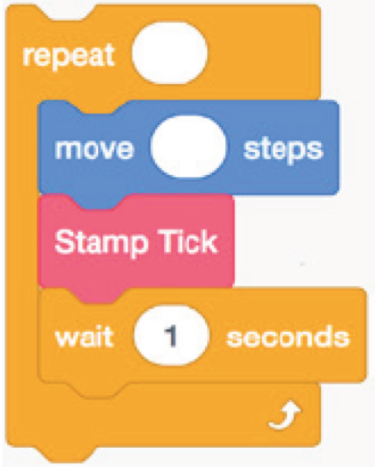
Debugging Activity: Student Page

④ Estimate what you think the numbers should be for fourths. Write them in the script on the right.



⑤ Check your estimates for Problem 4. If those numbers do not partition the line into four equal parts, then adjust, observe, and test until the output is four equal parts. Write those numbers in the script at the right.





Debugging Activity: Discussion Question(s)

Ask: *Did the program do exactly what you wanted it to do at first?* no *Did you just give up on the program and start over?* Sample answer: No, we modified the code by trying different values in the blocks until we got what we wanted. Explain to children that in CS this is called **debugging**: observing an outcome, finding any errors, adjusting the program, and testing the program to see if it produces the expected outcome. When programmers design a program, they test, observe, and adjust the program, then repeat these three steps until they have an expected result or the problem is solved. So in this case, children suggested an argument,

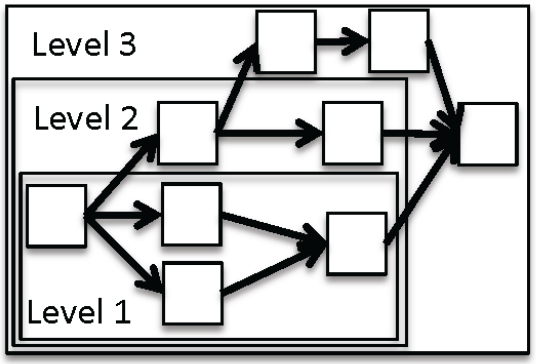
Outcomes can be used to decide whether or not there are errors.

→ Iterative refinement can help fix errors.

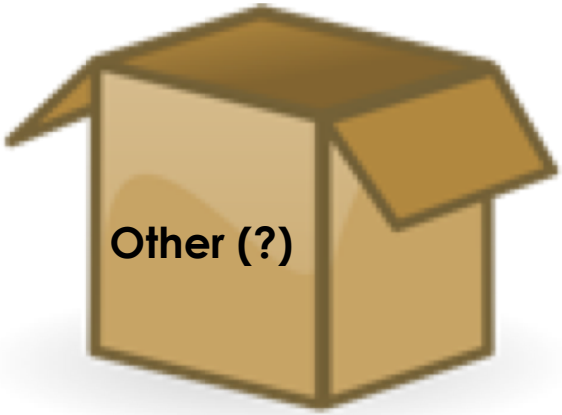
Collection of Consensus Goals (CGs)

Ordering

Learning Trajectory



- Our trajectories are nonlinear.
- Sorting by dimensions.



Dimension 1: Strategies for finding and fixing errors

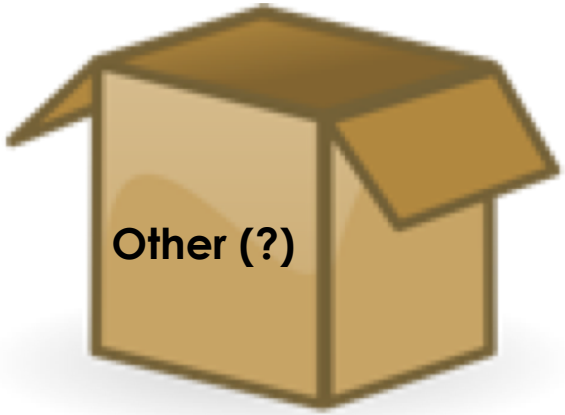
- This dimension collects *activities* of the practice.
- Strategies in the literature include:
  - Iterative refinement (trial & error)
  - Using intermediate results
  - Observing step-by-step execution
  - Reproducing errors
  - Addressing compile errors in order
- What is missing?

Dimension 2: Types of Errors

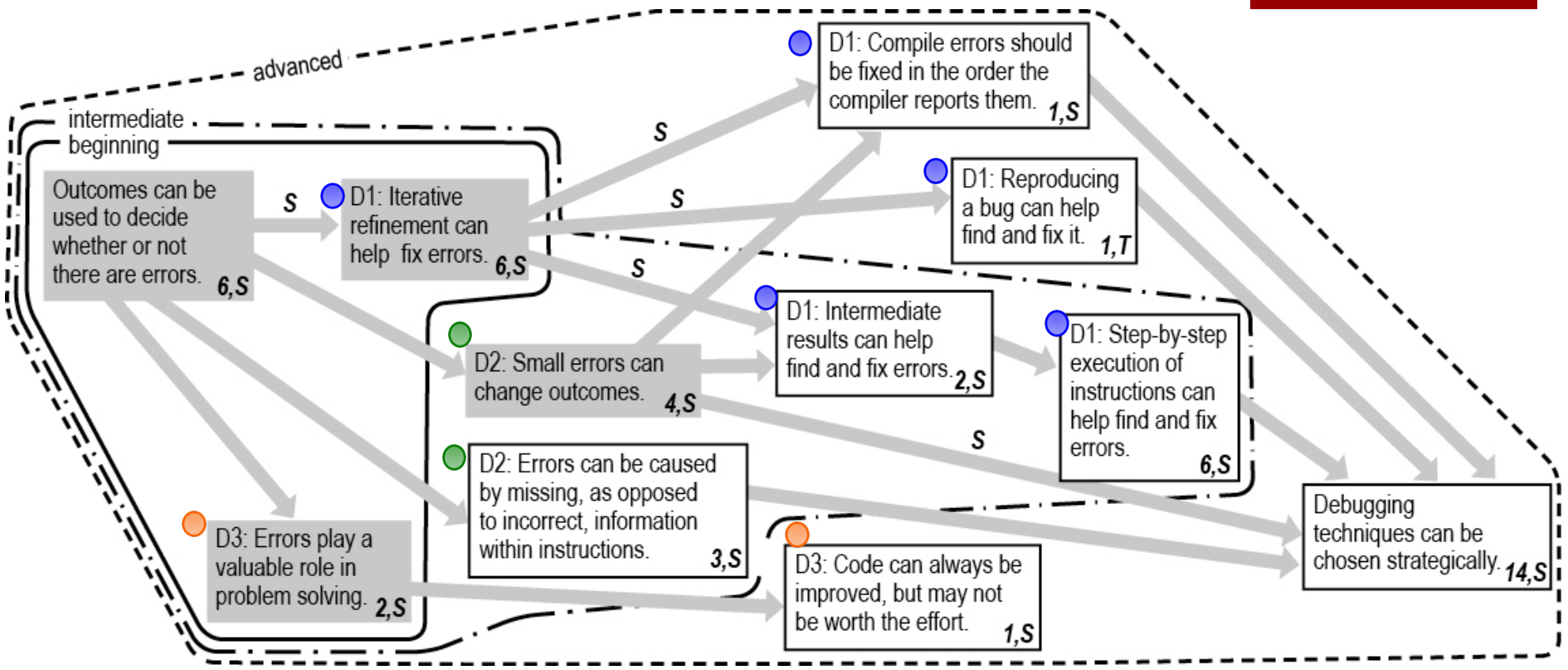
- This dimension collects *metaknowledge* of the practice.
- Error types in the literature include:
  - Small errors (e.g., case errors)
  - Errors of omission
- Again – what is missing?

Dimension 3: Role of Errors in Problem Solving

- This dimension is also *metaknowledge* of the practice, but includes an *affective* component.
- Consensus goals include:
  - Errors can be helpful.
  - Improvements can have diminishing returns.



The Debugging trajectory



Dimension 1: Strategies

Dimension 2: Types of Errors

Dimension 3: Role of Errors

UCHICAGO STEM EDUCATION

Thank you!
Contact Information

Katie Rich:
richkat3@msu.edu
@KatietheCurious

Carla Strickland:
castrickland@uchicago.edu
@CisforCarla

Diana Franklin:
dmfranklin@uchicago.edu

LTEC project website:
everydaycomputing.org

UCHICAGO STEM EDUCATION

A K–8 Debugging Learning Trajectory Derived from Research Literature

Kathryn M. Rich
Michigan State University
richkat3@msu.edu

Carla Strickland
UChicago STEM Education
castrickland@uchicago.edu

T. Andrew Binkowski
University of Chicago
abinkowski@uchicago.edu

Diana Franklin
UChicago STEM Education
dmfranklin@uchicago.edu

ABSTRACT

Curriculum development is dependent on the following question: What are the learning goals for a specific topic, and what are reasonable ways to organize and order those goals? Learning trajectories (LTs) for computational thinking (CT) topics will help to guide emerging curriculum development efforts for computer science in elementary school. This study describes the development of an LT for Debugging. We conducted a rigorous analysis of scholarly research on K–8 computer science education to extract what concepts in debugging students should and are capable of learning. The concepts were organized into the LT presented within. In this paper, we describe the three dimensions of debugging that emerged during the creation of the trajectory: (1) strategies for finding and fixing errors, (2) types of errors, and (3) the role of errors in problem solving. In doing so, we go beyond identification of specific debugging strategies to further articulate knowledge that would help students understand *when* to use those techniques and *why* they are successful. Finally, we illustrate how the Debugging LT has guided our efforts to develop an integrated mathematics and CT curriculum for grades 3–5.

KEYWORDS

Debugging, K–8, Learning trajectory, Computational thinking

ACM Reference Format:
Kathryn M. Rich, Carla Strickland, T. Andrew Binkowski, and Diana Franklin. 2019. A K–8 Debugging Learning Trajectory Derived from Research Literature. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE '19), February 27-March 2, 2019, Minneapolis, MN, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3287324.3287396

1 INTRODUCTION

Several school districts, including public schools in Chicago, New York City, and San Francisco, have begun CS for All initiatives that will bring computational thinking (CT) instruction to all students. The knowledge to create research-based curricula – namely what, how, and when to teach CT material – is still emerging. There exist two bodies of research literature to draw upon: one discussing what CT concepts students *should learn* [2–4, 23, 37] and the other exploring what students at different ages *did learn* through various CT-related activities [14–16, 19, 21, 22, 33] This paper brings these two bodies of work together to develop a learning trajectory (LT) for the CT practice of debugging. Our research questions are:

- What debugging concepts should children be taught?
- How could those concepts be organized?
- What are reasonable orders for addressing those concepts?

We build on prior work synthesizing existing research into guidance for practitioners and curriculum developers [10, 17, 18]. In particular, we make the following contributions:

- Identification of specific *learning goals* for debugging.
- Identification of three *dimensions* of debugging that can be used to organize those learning goals.
- Identification of literature that supports partial ordering of the debugging learning goals.

The rest of the paper is organized as follows. First, we describe related work. Next, we describe the methods we used to develop the trajectory and present our LT for Debugging. Then we share examples of how we have used the LT for curriculum development. We end with a conclusion and discussion of limitations.

2 RELATED WORK

We draw on three bodies of existing work: theory of learning trajectories; empirical research on debugging; and research and syntheses about CS in elementary school.

2.1 Learning Trajectories

The National Research Council [11] defines learning progressions as "descriptions of the successively more sophisticated ways of thinking about a topic that can follow one another as children learn about and investigate a topic" (p. 214). Typically, a learning progression starts with knowledge that students are expected to bring to a learning experience and ends with a societal norm of what students should know. Between these endpoints are intermediate stages representing progress toward the overall goal [5]. A less linear model, called Pieces of Knowledge [24], expresses independent kernels of knowledge that one could learn in any order, but that all must be learned to gain full understanding of a concept.

SIGCSE '19, February 27-March 2, 2019, Minneapolis, MN, USA

ACM ISBN 978-1-4503-5890-3/19/02…$15.00
https://doi.org/10.1145/3287324.3287396

A related construct is a *learning trajectory* (LT), which Simon [35] defines as a prediction of how learning might proceed via students' engagement in a particular activity. An LT has three parts: (1) an overall learning goal, (2) a predicted pathway to the learning goal (with periodic waypoints), and (3) instructional activities that help students move along the path [8]. A key difference between a learning progression and an LT is the latter's attention to activities [5]. This paper focuses on the first two components of our Debugging LT, goals and pathways, because full discussion of activities is beyond the scope of this paper. However, we acknowledge that the draft version of the LT presented here will both *shape* and *be shaped by* our ongoing curriculum development efforts. In the discussion, we share one activity developed through use of the LT. In future work, we will present results of tests of these activities and how we used the results of the studies to refine and elaborate the LT.

As we developed this Debugging LT, we drew, in particular, on the work of other researchers who have developed progressions for disciplinary practices in science. In Schwarz et al.'s [32] work on scientific modeling, the learning progressions are described not as a single sequence of levels of understanding, but as a set of dimensions of the practice. The dimensions of a practice are components that are related, but can develop separately. Often, dimensions separate metaknowledge of a practice from the activities of a practice. Schwarz et al.'s two dimensions of scientific modeling are *scientific models as tools for predicting and explaining* and *models change as understanding improves.* The first dimension roughly corresponds to metaknowledge about the practice, and the second to the activity of the practice. Following this work, we aimed to identify dimensions of debugging, with at least one focused on metaknowledge and at least one focused on activity . The dimensions that emerged from our analysis are presented in Section 4.

2.2 Debugging

In a seminal study, Klahr and Carver [26] identified four stages of debugging: Bug identification, or describing a discrepancy between intended and actual program outcome; Program representation, or thinking about the program structure to hypothesize possible locations for the bug; Bug location, or inspecting the program at hypothesized locations to search for errors; and Bug correction, or making the correction. In this paper, we refer to this as an Observe -> Hypothesize -> Modify -> Test cycle used to debug programs.

More recently, embodied pre-coding instruction in debugging was found to be helpful, improving students' confidence and abilities to cope with errors [1]. A debugging game, Gidget, was created for teens to learn debugging skills [27]. The environment scaffolded the process of debugging by identifying the existence of a bug.

2.3 CT Learning Trajectories

Researchers have attempted to discover what CT concepts students use successfully at different ages [14, 19, 34]. In addition, several efforts have synthesized existing knowledge and research to guide future curriculum development. The K-12 CS Framework [18] was released in spring 2016, the CSTA released standards soon afterward [17], and several states have used these documents to guide their own standards development. In addition, we recently published LTs for sequence, repetition, conditionals [31], and decomposition [30].

3 METHODS

Here we briefly describe our three phases of work, inspired by [10].

3.1 Identifying Learning Goals

We conducted a review of 108 articles [29] focused on computer-science education found in the Educational Research Information Center database, the Special Interest Group in Computer Science Education (SIGCSE) conference proceedings, and the Innovation and Technology in Computer Science Education (ITiCSE) conference proceedings using keywords such as "computational thinking," "computer science domains," and "computer science pedagogy," with the additional signifiers "K–8," "K–5," and "K–12." We focused on research published between 2006 and 2016. During this review, we extracted debugging-related learning goals (LGs) mentioned by the authors. For the purposes of this trajectory, a learning goal is defined as *any explicit statement or implicit endorsement of what students can or should* ***know*** *or be able to* ***do*** *in relation to debugging.* Attention to both knowing (metaknowledge) and doing (practice-based activity) was particularly important for learning goals related to debugging (a practice), as described in Section 2.1.

As it was extracted, each learning goal was tagged with an *evidence type:* student evidence if the goal was supported by work with students, and theoretical evidence if not.

3.2 Synthesizing Consensus Goals

We next synthesized small collections of LGs that expressed similar ideas into consensus goals (CGs). The four authors (two with computer science expertise and two with curriculum development expertise) created independent groupings, then came to collaborative agreement on the final set of CGs. Examples of groups of LGs that were synthesized into single CGs are included in Section 4.1.

A cross-check was performed on all consensus goals with relevant learning goals to ensure that the CGs aligned with the original intent of the LGs. Each CG was then coded with its strongest evidence type. That is, if at least one LG under a CG had student evidence, the CG was coded as having student evidence. Otherwise, it was coded as having theoretical evidence.

3.3 Connecting Consensus Goals

To begin organizing the CGs into pathways, we next identified dimensions of debugging by looking for threads of CGs that could be described independently of each other. Following the lead of [32], we looked, in particular, for at least one dimension focusing on metaknowledge (what students should *know*) and one focusing on activity (what students should be able to *do*). We then organized the CGs related to each dimension by increasing complexity. Ordering was based on evidence from the source articles, the theoretical assumption that trajectories should progress from knowledge familiar to the student toward formalized disciplinary knowledge, and consideration of hypothetical activities in which students might engage. Orders were represented as arrows between CGs. As part of this assembly process, we also identified suggested paths through the trajectory for beginning, intermediate, and advanced students. We acknowledge that our proposed pathways do not represent the only way a student could learn these concepts. Rather than functioning as a strict prescription, our arrows are intended to define possible

learning pathways that allow students to learn gradually, building on their prior knowledge. Additionally, we note that because much of the source literature involved students using block-based (rather than text-based) languages, our ordering of goals is most appropriate for students using graphical programming languages.

4 RESULTS

We now present the results of this process. We begin by delving into the development of consensus goals. We then present the learning trajectory itself with the description, literature evidence, and rationale for the dimensions and connections between the CGs.

4.1 Consensus Goals

Eleven CGs were synthesized from 36 LGs – 25 with student support and 11 with theoretical support. Table 1 presents each CG, along with a comment placing it in context, followed by one example LG that influenced that CG and the number of LGs that were synthesized into each CG.

To illustrate the process of synthesizing several LGs into one CG, we show two examples. These two examples are chosen because they illustrate the subtleties involved in creating CGs. The source literature was consulted often to glean the context of individual learning goals and remain true to their original intent.

Table 2 shows the three LGs that were synthesized into the CG, "Errors can be caused by missing, as opposed to incorrect, information within instructions." The evidence code for each goal is given in parentheses after the goal text. This table illustrates one way in which similarities between individual LGs were identified. Most of the debugging LGs referred in some way to fixing code. To create CGs, we gathered LGs that were similar in their description of either *what* was to be fixed or *how* it was to be fixed. LG1–LG3 in Table 2 are examples of the former: they all refer in some way to incompleteness of code. The presumed omission is *what* needs to be fixed. The CG therefore refers to the idea that errors can be caused by omission. These ideas of *what needs to be fixed* and *how it should be fixed* correspond to the types of errors (D2) and strategies (D1) dimensions of the LT described in Section 4.2.

Table 3 shows the six LGs that were synthesized into the CG, "Iterative refinement can fix errors." In contrast to the focus on *what* should be fixed in the CG in Table 2, this CG focuses on *how* errors could be fixed. Table 3 includes LGs that exemplify different aspects of the Observe -> Hypothesize -> Modify -> Test cycle of debugging based on outcome. These seemingly independent learning goals are tied together by a discussion within one paper [16] that describes different methods students used to fix their code, and the skills that led to different levels of success. This discussion of the different skills as part of a single process led to the CG and the identification of other LGs that address individual aspects of this cycle.

Note that the CGs are written to capture not the specific actions or skills cited in the individual LGs, but rather the knowledge that students need to perform those actions. We chose to word CGs in this manner for two reasons. First, we felt that wording goals in terms of understanding made the big ideas in the trajectory more transparent for more audiences (e.g., teachers with varying CS/CT experience). Second, understanding goals are less prescriptive than specific action goals, and we hope this allows the trajectories to be used flexibly for curriculum development.

4.2 Debugging Trajectory and Dimensions

Figure 1 depicts the Debugging LT. The four gray boxes are offline, or "unplugged," CGs, whereas the seven white boxes are computer-based CGs. A CG is marked as offline if the concept is general enough to be applied in a setting outside of a computer program. Information on the lower right corner of each box indicates the number of LGs that were coalesced into this consensus goal and the stronger type of evidence found for a CG (S for student evidence, T for theoretical evidence). The code in the upper left corner of most boxes (D1, D2, or D3) refers to the dimension on which the CG appears. The dimensions are described later in this section.

The CGs are connected by 16 arrows, 5 of which are supported by information extracted from the literature. The type of literature support found for an arrow is also indicated as student-supported (S) or theoretically-supported (T). Arrows without a letter label are based on theoretical considerations, rather than literature evidence.

The Debugging trajectory begins with a foundational skill for independent debugging: the ability to recognize whether or not there is an error. Literature suggests that without scaffolding (Section 5.1), this is a prerequisite to successful debugging. Flannery and Bers [16] described three groups of PK-K children programming a robot to dance. One group had trouble evaluating whether current solutions had problems or not. The two other, more advanced groups recognized errors and attempted to fix them via trial-and-error strategies, but had varying success. This evidence suggests that understanding that there is an *intended* outcome for most tasks, and being able to tell whether the actual outcome matches the intended outcome, is critical to successful debugging.

From this starting point, the consensus goals were organized into three dimensions, represented roughly from top to bottom in Figure 1: (D1) strategies for finding and fixing errors, (D2) types of errors, and (D3) the role of errors in problem solving. Dimension 1 largely focuses on activity, while dimensions 2 and 3 largely focus on the metaknowledge that makes debugging activity meaningful, akin to prior work in science [32].

4.2.1 D1: Strategies for finding and fixing errors. The first debugging dimension (D1), *strategies for finding and fixing errors,* consists of five debugging strategies discussed in the literature: iterative refinement by trial and error, using intermediate results, observing step-by-step execution, reproducing errors, and addressing compile errors in order of appearance.

Research evidence suggests trial and error is the first debugging strategy attempted by students. Both Flannery and Bers [16] and Fessakis, Gouli, and Mavroudi [15] found that PK-K children utilized trial and error as their debugging strategy. In addition, trial and error is the most direct use of the observed incorrect outcome, taking the recognition of a bug a step farther to acknowledge that the particular nature of the unexpected outcome can be used to theorize what the problem might be, make a change, and check if the problem is fixed. Thus, iterative refinement is placed immediately after the recognition of errors in the beginning strand, with the trial and error strategy being the simplest form of iterative refinement.

Table 1: The 11 Debugging Consensus Goals

Consensus Goal	Comments	Example Supporting LG	No. of LGs
Outcomes can be used to decide whether or not there are errors.	Realizing there is an error is the first step to fixing it, and critically analyzing the outcome is a way to do it.	"Recognize when instructions do not correspond to actions." [2]	6
Iterative refinement can help fix errors.	This is the first step toward understanding that Observe -> Hypothesize -> Modify -> Test [26] can be used to debug code.	Make a hypothesis about the cause of a problem. [6]	6
Errors play a valuable role in problem solving.	This CG is particularly useful for young children who need to manage emotions. Errors are normal and useful.	"[D]ebug understandings if their instructions produce something unexpected"	2
Small errors can change outcomes.	Students should understand that programming errors are often small (case changes, number of iterations), despite their potentially large effects.	"[U]nderstand that small errors" can change outcomes. [13]	4
Intermediate results can help find and fix errors.	This CG progresses from the idea of looking at the final result to recognize errors. Students can also examine intermediate results.	Fix nearly complete code by "inserting wait blocks" to see intermediate results. [33]	2
Step-by-step execution of instructions can help find and fix errors.	This technique builds on the usefulness of intermediate results.	"Desk-check a solution" by tracing by hand. [20]	6
Errors can be caused by missing, as opposed to incorrect, information within instructions.	Distinguishing between ordering of instructions, wrong arguments, and the absence of necessary instructions.	"[D]ebug" or complete an existing program. [15]	3
Reproducing a bug can help find and fix it.	This is advanced for K–8. The difference between reproducing a bug and rerunning a program is subtle for young learners.	Reproduce an unexpected problem. [36]	1
Compile errors should be fixed in the order the compiler reports them.	This CG is also advanced, as it is most appropriate for text-based languages.	"[F]ix a program that had errors" identified by the parser. [21]	1
Code can always be improved, but it may not be worth the effort.	This CG serves as a bridge between debugging and quality – there is a fine line between bugs and elements that could be improved (e.g., speed).	"[M]anage their programming processes" by questioning "their decisions and actions" to reflect on debugging processes. [25]	1
Debugging techniques can be chosen strategically.	This is the capstone CG for debugging. Once students understand individual techniques, they should choose between them strategically.	Correct errors in a "systematic, efficient manner" [36]	14

Table 2: Synthesis Example 1

LG1	Fix nearly complete code. [33] (S)
LG2	Identify "what [is]lacking in instructions." [14] (S)
LG3	"[D]ebug" or complete an existing program. [15] (S)
CG	Errors can be caused by missing, as opposed to incorrect, information within instructions. (S)

Research literature did not provide insight into the relationship between the remaining four debugging strategies, though there is a similarity between the use of intermediate results and step-by-step execution. Namely, the state of a program after each step is executed is, in itself, an intermediate result. Thus, these two strategies are linearly connected in the LT. Using intermediate results is located

Table 3: Synthesis Example 2

LG1	Attempt to solve a problem within a program. [16] (S)
LG2	Use "a trial and error model" to solve a problem. [15] (S)
LG3	Make a hypothesis about the cause of a problem. [16] (S)
LG4	Test a theory about what is causing a problem in a program. [27] (T)
LG5	Decide how to change a program when it does not produce the desired results. [7] (S)
LG6	"Repeatedly change and run" buggy "code to find a solution" [6] (S)
CG	Iterative refinement can help fix errors. (S)

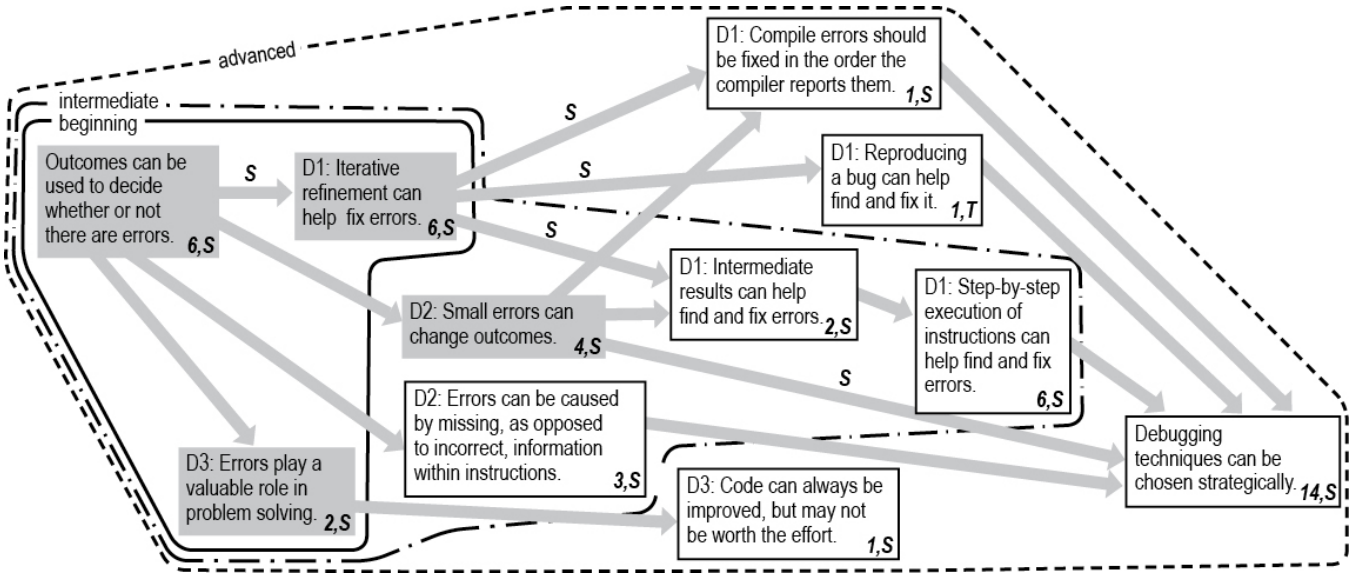


Figure 1: The Debugging Trajectory

prior to step-by-step execution because step-by-step execution is a particular strategy using intermediate results, whereas the notion of using intermediate results could lead generally to other strategies (such as removing the last half of the code and running it to see if the bug is in the first half or last half of that script or function). The other two strategies, reproducing bugs and addressing compiler errors in order, are not related and so are located on separate paths.

Using intermediate results and step-by-step execution are considered intermediate concepts because they require more deliberate thought than trial and error, but they are not considered advanced because students can use them in any non-trivial program. Reproducing an error is considered advanced because the need to reproduce an error implies that the error does not necessarily always occur, and that step-by-step execution may not be a successful strategy. This fact alone makes such bugs more difficult to identify and fix than others. Addressing compile errors is considered advanced because students are more likely to begin with languages such as Scratch [28] that are designed to avoid compile errors.

4.2.2 D2: Types of errors. The second dimension (D2), *types of errors,* contains two types of errors: small errors, which may include errors in punctuation, syntax, and case, and errors of omission, which may include missing instructions or use of instructions that lack the appropriate precision to communicate needed information. There are fundamental differences in these two ideas. For example, the cognitive act of scrutinizing existing code to find a small error is different from the cognitive act of discerning missing information or steps. However, the research we reviewed did not provide evidence of any dependence between these ideas, so they exist independent paths in the LT. These CGs are in the intermediate strand, as opposed to beginner, because the trial-and-error debugging strategies used by beginners do not suggest that novice students attend to the differences in errors. They are on the intermediate path, as opposed to advanced, because each was supported by at least one LG that stemmed from work with students in grades 3–5 [14, 21, 33].

4.2.3 Role of errors in problem solving. The third dimension (D3), *the role of errors in problem solving*, consists of only two CGs: understanding the value of errors in problem solving and assessing the worth of improvements. The former is in the beginning strand, as we consider this a powerful and pervasive idea that can be taught in elementary classrooms outside of computing instruction. Young children have trouble regulating their emotions, making it hard to persevere when they encounter failure [12], so explicitly teaching the prevalence and usefulness of errors could help young learners the most, both generally and in the context of programming.

The latter CG is in the advanced strand and meant to address the close relationship between fixing errors and making improvements in quality and how to assess whether improvements are worthwhile. Very simple programs may have a "right answer," giving a programmer a point at which to stop debugging. However, more complex programs have many design decisions that have no single, clear solution, and there becomes a subtle difference between a bug and a desire to improve (e.g., improve performance or make the interface easier to use).

The trajectory ends with the consensus goal "Debugging techniques can be chosen strategically." This nexus is intended to capture the idea that one must deliberately choose a strategy based on the problem at hand. This ability to choose between debugging strategies and approaches is a more complex skill than learning the individual ideas or strategies.

5 DISCUSSION

This trajectory is only a starting point. An LT consists of more than the one-sentence consensus goals described in this paper and suggested paths through them. Full details of our LT can be found online [9]. Each consensus goal has an understanding goal

(described in this paper) and an action goal. In addition, there are example activities that move students from one consensus goal to the next, building on the prior knowledge.

LTs can be used in several ways. They can serve as the basis of experiments to verify or refute connections within the LT. However, this LT's most promising role is in curriculum development. We first discuss its use in creating scaffolding that can allow curricula to temporarily bypass specific CGs. Second, we discuss an example of how it was used to create a lesson in an integrated mathematics + CT curriculum in order to explicitly teach debugging.

5.1 Guiding Scaffolding

The ordering within the learning trajectory is not intended to be entirely prescriptive. Indeed, scaffolding and careful activity design can allow for the consensus goals to be addressed in a different order. For example, although the ability to recognize the presence an error is listed first in the Debugging LT, a system or activity in which the identification of the error is provided would allow students to learn and practice some debugging skills without requiring the ability to recognize or identify the specific errors (e.g., Gidget [27]). In this case, however, the learning trajectory is very useful in guiding what specific things must be provided by scaffolding in order to allow students to be successful in later learning goals.

5.2 Guiding Activity Development

The LT also can guide activity development. The authors are developing an integrated mathematics + CT curriculum guided by this and other LTs. This interdisciplinary endeavor began with the identification of "anchor" activities in which CT concepts can be used to support mathematics learning. The LTs were then used to identify what necessary CT skills should be developed in order to prepare students for the anchor activities. Activities were designed to build the identified skills.

Here, we describe one activity for elementary school students, developed with reference to the Debugging LT, but not yet published by the authors. This activity is designed to explicitly teach students to closely observe program execution and think critically about what happened, corresponding to the consensus goals "Outcomes can be used to decide whether or not there are errors." and "Iterative refinement can help fix errors." While the latter goal can encompass varying degrees of reasoning (from random changes to well-thought-out changes), we are explicitly teaching students to carry out the process in a methodical manner.

We designed this activity to teach debugging and repeat loops within the mathematical context of representing fractions on a number line. The primary task is to modify code in Scratch that partitions a number line, so that a number line segment is divided into four equal parts. The starting code uses move and stamp blocks to divide a number line segment covering the interval 0 to 360 into four partitions of 50 units (with 160 left over). Students run the program, observe, and record the outcome. They are asked to identify what is wrong (partitions are too small, entire interval is not partitioned), what block affected that (move), and whether the current number of steps used as an argument in the move block needs to be made smaller or bigger (bigger). They are then prompted to choose a new number for the argument and repeat the process until they find the correct number to accomplish the four equal partitions (for this age group, large-number division is not expected). Students are then asked to repeat the process of finding the proper arguments to partition the number line into 6 equal parts, adjusting both the number of iterations in the repeat loop and the number of steps in the move block.

The cognitive process of reasoning about what was wrong, shown in our Debugging LT, is discernable in this activity. Once students observe and identify the problem (the parts are too small), they then identify the block that caused it, hypothesize about what direction they should take in fixing it, and make slight modifications to the program to test their hypotheses. The process of debugging the program is broken down into discrete steps (Observe >Hypothesize->Modify->Test) that even young children can handle. In future studies, we plan to use activities such as this one to test whether, for small children, making these steps explicit can help them be successful in future debugging efforts. Preliminary field testing has shown that 3rd and 4th grade students enjoy and successfully engage with this activity in an elementary mathematics classroom setting. Students even noted similarities with hypothesizing and testing in science and suggested that they debug every time they fix a problem, not just a program.

6 CONCLUSIONS AND LIMITATIONS

Here we present a learning trajectory related to the practice of Debugging in CS/CT and present a description of the dimensions of debugging represented in the trajectory. While the current version of the Debugging trajectory includes all the particular strategies and error types that were mentioned in the reviewed literature, we do not imagine that these are a definitive list of what might be taught in elementary CT curricula. As we reviewed the existing CS/CT literature, we found the topic of debugging to be underrepresented in comparison to other topics such as loops and conditionals. As debugging is inherent to all practices of CS/CT, it is possible that it has been overlooked or unrecognized as a separate concept, especially during early introductions to CS/CT, where students may expend more effort debugging than implementing newly learned concepts. We hope that this trajectory signifies a more isolated and targeted view of debugging practices in early CS/CT education.

Limitations of this LT are rooted in our specific focus on K-8 students and the dominance of block-based languages in the reviewed literature. It is likely that the grade level and specific language and platform will influence student debugging practices. This is especially interesting as students transition from block-based languages, which are purposefully designed to prevent certain types of errors, to text-based languages that will introduce students to a new class of errors and debugging strategies. We expect that additional strategies and types of errors will prove useful and merit inclusion in this trajectory as more research is added to the field.

ACKNOWLEDGMENTS

This material is based on work supported by the National Science Foundation under Award 1542828 and Award 1742466. Any opinions, findings, and conclusions or recommendations expressed are those of the authors and do not necessarily reflect those of the National Science Foundation.

REFERENCES

[1] Jung-Hyun Ahn, Yaoli Mao, Woonhee Sung, and John B Black. 2017. Supporting Debugging Skills: Using Embodied Instructions in ChildrenâĂŹs Programming Education. In *Society for Information Technology & Teacher Education International Conference*. Association for the Advancement of Computing in Education (AACE), 19–26.

[2] Charoula Angeli, Joke Voogt, Andrew Fluck, Mary Webb, Margaret Cox, Joyce Malyn-Smith, and Jason Zagami. 2016. A K-6 computational thinking curriculum framework: Implications for teacher knowledge. *Journal of Educational Technology & Society* 19, 3 (2016).

[3] Michal Armoni and Judith Gal-Ezer. 2014. Early computing education: why? what? when? who? *ACM Inroads* 5, 4 (2014), 54–59.

[4] Valerie Barr and Chris Stephenson. 2011. Bringing computational thinking to K-12: what is Involved and what is the role of the computer science education community? *Acm Inroads* 2, 1 (2011), 48–54.

[5] Michael T Battista. 2011. Conceptualizations and issues related to learning progressions, learning trajectories, and levels of sophistication. *The Mathematics Enthusiast* 8, 3 (2011), 507–570.

[6] Matthew Berland, Taylor Martin, Tom Benton, Carmen Petrick Smith, and Don Davis. 2013. Using learning analytics to understand the learning pathways of novice programmers. *Journal of the Learning Sciences* 22, 4 (2013), 564–599.

[7] Douglas H Clements. 2002. Computers in early childhood mathematics. *Contemporary issues in early childhood* 3, 2 (2002), 160–181.

[8] Douglas H Clements and Julie Sarama. 2004. Learning trajectories in mathematics education. *Mathematical thinking and learning* 6, 2 (2004), 81–89.

[9] Everyday Computing. 2019. Trajectory Visualizations. Retrieved February 27, 2019 from http://everydaycomputing.org/public/visualization/

[10] Jere Confrey, Alan P Maloney, and Andrew K Corley. 2014. Learning trajectories: a framework for connecting standards with curriculum. *ZDM* 46, 5 (2014), 719–733.

[11] National Research Council et al. 2007. *Taking science to school: Learning and teaching science in grades K-8*. National Academies Press.

[12] Adele Diamond and Kathleen Lee. 2011. Interventions shown to aid executive function development in children 4 to 12 years old. *Science* 333, 6045 (2011), 959–964.

[13] Hilary Dwyer, Bryce Boe, Charlotte Hill, Diana Franklin, and Danielle Harlow. 2013. Computational Thinking for Physics: Programming Models of Physics Phenomenon in Elementary School. *Engelhardt, Churukian, & Jones (Eds.) 2013 PERC Proceedings* (2013), 133–136.

[14] Hilary Dwyer, Charlotte Hill, Stacey Carpenter, Danielle Harlow, and Diana Franklin. 2014. Identifying elementary students' pre-instructional ability to develop algorithms and step-by-step instructions. In *Proceedings of the 45th ACM technical symposium on Computer science education*. ACM, 511–516.

[15] Georgios Fessakis, Evangelia Gouli, and E Mavroudi. 2013. Problem solving by 5–6 years old kindergarten children in a computer programming environment: A case study. *Computers & Education* 63 (2013), 87–97.

[16] Louise P Flannery and Marina Umaschi Bers. 2013. LetâĂŹs dance the âĂIJrobot hokey-pokey!âĂİ ChildrenâĂŹs programming approaches and achievement throughout early cognitive development. *Journal of research on technology in education* 46, 1 (2013), 81–101.

[17] CSTA Standards Task Force. 2011. *CSTA K–12 Computer Science Standards*. Technical Report. Computer Science Teacher's Association, New York.

[18] K-12 Computer Science Framework. 2016. K–12 Computer Science Framework. Retrieved October 10, 2018 from http://k12cs.org

[19] Diana Franklin, Gabriela Skifstad, Reiny Rolock, Isha Mehrotra, Valerie Ding, Alexandria Hansen, David Weintrop, and Danielle Harlow. 2017. Using upper-elementary student performance to understand conceptual sequencing in a blocks-based curriculum. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. ACM, 231–236.

[20] Ursula Fuller, Colin G Johnson, Tuukka Ahoniemi, Diana Cukierman, Isidoro Hernán-Losada, Jana Jackova, Essi Lahtinen, Tracy L Lewis, Donna McGee Thompson, Charles Riedesel, et al. 2007. Developing a computer science-specific learning taxonomy. *ACM SIGCSE Bulletin* 39, 4 (2007), 152–170.

[21] Chris Gregg, Luther Tychonievich, James Cohoon, and Kim Hazelwood. 2012. EcoSim: a language and experience teaching parallel programming in elementary school. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education*. ACM, 51–56.

[22] Shuchi Grover and Satabdi Basu. 2017. Measuring student learning in introductory block-based programming: Examining misconceptions of loops, variables, and boolean logic. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. ACM, 267–272.

[23] Shuchi Grover and Roy Pea. 2013. Computational thinking in K–12: A review of the state of the field. *Educational Researcher* 42, 1 (2013), 38–43.

[24] David Hammer and Tiffany-Rose Sikorski. 2015. Implications of complexity for research on learning progressions. *Science Education* 99, 3 (2015), 424–431.

[25] Filiz Kalelioğlu. 2015. A new way of teaching programming skills to K-12 students: Code. org. *Computers in Human Behavior* 52 (2015), 200–210.

[26] David Klahr and Sharon McCoy Carver. 1988. Cognitive objectives in a LOGO debugging curriculum: Instruction, learning, and transfer. *Cognitive Psychology* 20, 3 (1988), 362–404.

[27] Michael J Lee, Faezeh Bahmani, Irwin Kwan, Jilian LaFerte, Polina Charters, Amber Horvath, Fanny Luor, Jill Cao, Catherine Law, Michael Beswetherick, et al. 2014. Principles of a debugging-first puzzle game for computing education. In *Visual Languages and Human-Centric Computing (VL/HCC), 2014 IEEE Symposium on*. IEEE, 57–64.

[28] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. 2010. The scratch programming language and environment. *ACM Transactions on Computing Education (TOCE)* 10, 4 (2010), 16.

[29] Kathryn Rich, Carla Strickland, and Diana Franklin. 2017. A Literature Review through the Lens of Computer Science Learning Goals Theorized and Explored in Research. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. ACM, 495–500.

[30] Kathryn M Rich, T Andrew Binkowski, Carla Strickland, and Diana Franklin. 2018. Decomposition: A K-8 Computational Thinking Learning Trajectory. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*. ACM, 124–132.

[31] Kathryn M Rich, Carla Strickland, T Andrew Binkowski, Cheryl Moran, and Diana Franklin. 2018. K–8 learning trajectories derived from research literature: sequence, repetition, conditionals. *ACM Inroads* 9, 1 (2018), 46–55.

[32] Christina V. Schwarz, Brian J. Reiser, Elizabeth A. Davis, Lisa Kenyon, Andres Acher, David Fortus, Yael Shwartz, Barbara Hug, and Joe Krajcik. 2009. Developing a learning progression for scientific modeling: Making scientific modeling accessible and meaningful for learners. *Journal of Research in Science Teaching* 46, 6 (2009), 632–645.

[33] Linda Seiter. 2015. Using SOLO to classify the programming responses of primary grade students. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. ACM, 540–545.

[34] Linda Seiter and B. Foreman. 2013. Modeling the learning progressions of computational thinking of primary grade students. In *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research*. ACM, 59–66.

[35] Martin A. Simon. 1995. Reconstructing mathematics pedagogy from a constructivist perspective. *Journal for Research in Mathematics Education* 1995 (1995), 114–145.

[36] David Weintrop, Elham Beheshti, Michael Horn, Kai Orton, Kemi Jona, Laura Trouille, and Uri Wilensky. 2016. Defining computational thinking for mathematics and science classrooms. *Journal of Science Education and Technology* 25, 1 (2016), 127–147.

[37] Jeannette M. Wing. 2006. Computational thinking. *Commun. ACM* 49, 3 (2006), 33–36.