Multilingual CS Education Pathways: Implications for Vertically-Scaled Assessment

Yvonne Kao
WestEd
Redwood City, CA, USA
ykao@wested.org

David Weintrop
University of Maryland
College Park, MD, USA
weintrop@umd.edu

ABSTRACT

The expansion of computer science (CS) into K-12 contexts has resulted in a diverse ecosystem of curricula designed for various grade levels, teaching a variety of concepts, and using a wide array of different programming languages and environments. Many students will learn more than one programming language over the course of their studies. There is a growing need for computer science assessment that can measure student learning over time, but the multilingual learning pathways create two challenges for assessment in computer science. First, there are not validated assessments for all of the programming languages used in CS classrooms. Second, it is difficult to measure growth in student understanding over time when students move between programming languages as they progress in their CS education. In this position paper, we argue that the field of computing education research needs to develop methods and tools to better measure students' learning over time and across the different programming languages they learn along the way. In presenting this position, we share data that shows students approach assessment problems differently depending on the programming language, even when the problems are conceptually isomorphic, and discuss some approaches for developing multilingual assessments of student learning over time.

CCS CONCEPTS

• **Social and professional topics → K-12 education**; **Computer science education**; **Student assessment**.

KEYWORDS

computer science education, K-12 education, assessment, assessment validation, programming

ACM Reference Format:
Yvonne Kao and David Weintrop. 2022. Multilingual CS Education Pathways: Implications for Vertically-Scaled Assessment. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2022), March 3–5, 2022, Providence, RI, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3478431.3499315

SIGCSE 2022, March 3–5, 2022, Providence, RI, USA

ACM ISBN 978-1-4503-9070-5/22/03…$15.00
https://doi.org/10.1145/3478431.3499315

1 INTRODUCTION

Computer science (CS) is currently in a period of rapid expansion in K-12 classrooms. This growth is the result of work at every level of education, including national efforts to change policy and create standards and framework documents, concerted state and regional efforts to bring CS into classrooms, and individual districts, schools, and educators working to create opportunities for students to learn CS. To realize the vision of CS for All, the field still needs to make progress along a number of fronts, including assessment. School districts across the United States and national efforts in countries like the United Kingdom have invested significantly in launching and growing computer science pathways, beginning by implementing high school courses and extending down into middle and elementary schools. However, it has proved challenging to understand the impact of these efforts on students' computer science knowledge and researchers are calling for a system of assessments that can measure student learning over time [31].

Students will likely learn more than one programming language over the course of their computer science educational experiences. Existing K-12 computer science pathways will generally expose students to at least two programming languages and possibly many more. Early learning experiences are likely to incorporate block-based programming environments like Scratch, which is used by a growing number of curricula (e.g. MyCS [27], Scratch Encore [14], and Exploring Computer Science [29]). Later K-12 instruction often focuses on text-based programming languages like Java or Python. Given the variety of languages and programming environments currently in use in K-12 education, measuring the growth of students as they progress through a CS education pathway is an open and important question.

In this position paper, we argue for the need to develop more sophisticated and rigorous ways to track the growth of students as they progress through a multi-year disciplinary pathway that may span multiple programming languages. Following the lead of other disciplines, in this work we argue for the use of vertically-scaled assessments. These are systems of assessments where scores from tests given at different time points can be placed on the same scale to estimate how much a student has learned between assessment administrations. As an example, the Smarter Balanced Assessment Consortium in the United States has created vertically-scaled assessments in mathematics and English language arts. Students take a Smarter Balanced assessment each spring and their scores are then compared year-over-year to measure growth [7].

Creating vertically-scaled assessments in computer science is challenging, in part because of the diversity of programming languages used in CS education. Consider the following scenario. A student takes a first computer science course using a curriculum

written for the block-based programming language Scratch. The following year, the student takes a second, more advanced course, this time taught in the text-based programming language Python. How much did the student learn about core CS concepts in the Python course, *above and beyond* what was learned in the Scratch course?

There are a few ways to answer this question. We could administer a Scratch assessment at the end of each course and compare the scores. This solution is suboptimal, as the student may be out of practice with Scratch and may also have learned new CS concepts in Python that the student has never implemented in Scratch. This would also render us unable to assess student learning on Python concepts that have no clear equivalent in Scratch. Alternatively, we could administer a Scratch assessment at the end of the first course, and administer a Python assessment at the end of the second course that has some items translated from Scratch to Python and compare the student's performance on the translated items.

Typically, vertically-scaled assessments are designed with items that appear on multiple test forms to help establish the common scale. Creating vertically-scaled assessments in computer science has the unique challenge of needing to assess the same concepts in multiple programming languages. However, the data we present in this paper suggests that students may not perform identically when items are presented in different programming languages, even when the items are conceptually the same.

Whether or not a course pathway is successful in advancing student knowledge over time is an important empirical question for evaluating and improving K-12 CS programs. This position paper seeks to encourage discussion and debate on what a vertically-scaled assessment in computer science might look like, how multilingual CS education pathways impact how we design and interpret the scores from such assessments, and what additional work needs to be done to create and validate them.

This paper continues with a brief review of the literature on multilingual CS education and assessment. We then present data from an assessment that we translated into three different programming languages and then administered in three different high school classroom contexts. The findings section presents overall assessment results by course as well as a more detailed look at student performance on specific items by course. The paper concludes with a discussion of implications of this work and future work that would be needed to develop an assessment that could measure growth in student learning over time and through different programming languages.

2 PRIOR WORK

2.1 Transferring Knowledge Between Programming Languages

The effects of transitioning students from one language to another is an area of active study. Work from the 1990s that examined transfer between programming languages found that this process often does not go smoothly [28]. Although there is a body of conceptual knowledge that applies across programming languages, there can be little enough overlap in syntax and style between previously-learned and new languages that code-writing skill in the new language must be developed essentially from scratch [1].

More recently, there has been a significant amount of research following learners in moving from an introductory block-based context to more conventional text-based programming languages. Cliburn reported a study of undergraduate students who first learned to program in the Alice environment and then transitioned to Java. Less than 60% of students felt the earlier experience in Alice was helpful for learning Java. This finding discouraged the author's continued use of Alice [6]. Armoni et al. found mixed evidence for transfer of student learning from Scratch to C# or Java. While teachers reported that students with Scratch experience learned C# or Java more quickly, there was little difference in final assessment performance between students who had previously learned Scratch and those who had not [3]. Weintrop and Wilensky conducted a quasi-experimental study on the impact of introductory block-based instruction on students' learning of Java. They found that the block-based introduction neither impeded nor facilitated learning in Java. Although students who first studied a block-based language learned more quickly at the outset, they ultimately performed no better on an end-of-program Java assessment than students who did not first learn a block-based language [34]. Other studies have found positive effects of first introducing students to a block-based language and then supporting the transition to a text-based language with explicit scaffolds in place for the transition [2, 9, 20].

2.2 Assessing Computer Science Knowledge

The design and validation of computer science content assessments is an active area of research. Decker and McGill [10] conducted a comprehensive review of 47 instruments published between 2012 and 2016 to measure cognitive, non-cognitive, or program assessment constructs in CS education. Most (66%) of these instruments measured non-cognitive constructs (e.g., self-efficacy or sense of belonging) while 28% measured cognitive constructs (e.g., computational thinking or CS conceptual knowledge). Much of the work has been targeted at the undergraduate level [8, 12, 13, 22, 24], but they did find a smaller number of assessments at the elementary [5], middle [35] and high-school levels [33]. A few additional assessments have been published since Decker and McGill's review [23, 25, 36].

While many assessments rely on a specific programming language, there have been some efforts to create assessments of computer science knowledge that do not rely on knowledge of any particular programming language. For example, the current Advanced Placement (AP) CS Principles (CSP) course offered in the United States assesses students using a custom pseudocode and poses questions in both block-based and text-based forms of that pseudocode. This design has been found to benefit students from historically excluded populations in CS [32]. However, research on (and with) so-called language-independent assessments [12, 22] has identified challenges and drawbacks [16]. One major issue is that pseudocode-based assessments are not truly language-neutral or language-independent. The syntax and semantics of students' primary programming language influences the way they respond to pseudocode-based assessments. Students whose primary programming language is syntactically more similar to the assessment pseudocode also tend to perform better on the assessment [11, 33].

2.3 Summary

Taken together, the research on transfer between programming languages and multilingual assessment present a conundrum: there is sufficient transfer of knowledge from a first programming language to affect student performance on pseudocode-based assessments. However, there is not necessarily enough transfer from one language to another that students can pick up learning new concepts from where they left off if their previous coursework used a different language. This presents challenges for precisely measuring changes in student learning over longer periods of time. In the next section, we present a secondary data analysis that illustrates how language can influence high school students' responses on conceptually-equivalent assessment items.

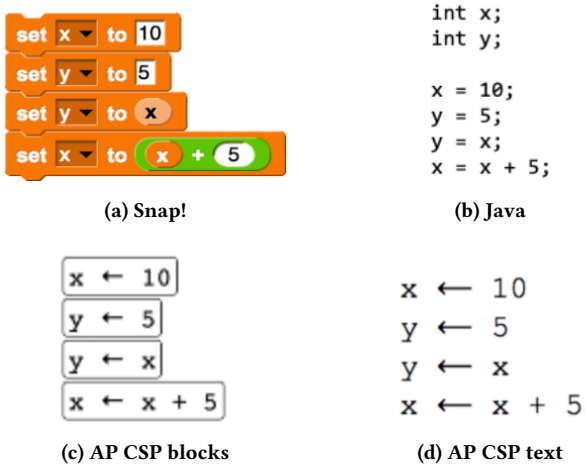
3 AN EXAMPLE FROM HIGH SCHOOL

3.1 The Assessment

This paper uses a subset of questions from the Commutative Assessment, a 28-question multiple choice assessment designed to evaluate learners' conceptual understanding of computing concepts across block-based and text-based modalities [33]. To measure student conceptual understanding independent of programming language and modality, the questions were designed such that students are asked to interpret short pieces of code that can be presented in either a block-based or text-based form. Every question presents the student with a short piece of code (either block-based or text-based) thus allowing the administrator of the assessment to gain insight into the relationship between how the program is presented (including language and modality) and student understanding. Further, this flexibility makes the assessment suitable for classes taught with either text-based or block-based languages, meaning it can be administered at multiple points along a multi-year CS pathway. A side-effect of this design is that none of the programs or concepts included in the assessment rely on a construct unique to any of the languages used. So for example, there are no questions that use Scratch blocks related to motion or costumes that do not have an analog in Java. At the same time, there are no questions related to type casting in Java as Scratch is a weakly-typed language. Figure 1 presents an example of a code snippet from a question on the Commutative Assessment presented in Snap! (Figure 1a), Java (Figure 1b), the AP CSP block-based pseudocode (Figure 1c), and the AP CSP text-based pseudocode (Figure 1d). For this question, students are asked: *What will be the value of x and y after this script is run?*

The multiple-choice options presented to the learner include the correct answer alongside distractor answers informed by prior research on programming misconceptions [30]. Importantly, the specific wording of the question and the multiple choice options are held constant across the different forms of the Commutative Assessment. The only thing that differs is the language and/or modality of the code snippet presented within each question.

This paper argues for the need for assessment approaches that can live at various points along a multi-year CS pathway. In the next section, we present an analysis focused on eight items from the Commutative Assessment, which we will collectively refer to as the Mini-CA, as a demonstration of the potential and pitfalls of using these conceptually isomorphic items in a multilingual vertically-scaled assessment. The Mini-CA is comprised of two



(a) Snap! (b) Java

(c) AP CSP blocks (d) AP CSP text

Figure 1: The code snippet for item Variables 2, presented in (a) Snap!, (b) Java, (c) the AP CSP block-based pseudocode, and (d) the AP CSP text-based pseudocode.

Table 1: Summary of student demographics by course.

Course	n	% Female	% URG	% Jrs or Srs
Intro to CS	177	28.3%	20.3%	67.2%
AP CSP	5156	27.4%	31.8%	82.4%
AP CS A	172	26.8%	8.7%	89.3%

items for each of four concepts: variables, conditionals, loops, and code comprehension (i.e., identifying the function or purpose of a chunk of code).

3.2 Participants

The data presented below are drawn from two different studies that included three high school contexts and three sets of participants: students enrolled in a pre-AP introductory CS course, AP CSP, and AP CS A. The three contexts are described in greater detail in the next section and reflect different stages in common high school CS course pathways in the United States. Table 1 summarizes the student demographics for each course. Most students identified as male, not as members of historically underrepresented racial or ethnic groups (URG), and as juniors or seniors in high school.

3.3 Course and Assessment Contexts

3.3.1 Introduction to Computer Science. The pre-AP Introduction to CS course used in this study was based on the Beauty and Joy of Computing (BJC) course developed by the University of California—Berkeley for non-CS majors and adapted for high school [15]. The course uses the Snap! programming environment, which is a block-based environment inspired by Scratch and includes additional features such as first-order functions [17]. The curriculum contains a variety of lessons and activities to teach programming fundamentals as well as lessons that address the applications and ethical implications of computing in modern society.

Table 2: Summary of student performance and Cronbach's alpha by course.

	% Correct		
Course	Mean	Std. Dev.	Cronbach's alpha
Intro to CS	75.1	28.3	0.82
AP CSP	83.4	23.3	0.79
AP CS A	70.4	27.1	0.75

3.3.2 AP CS Principles. The AP CS Principles course was designed to serve as an introduction to the field of computer science with a focus on framing the field as more than just programming, covering additional topics including algorithms, design, data, and social impacts of computing. The AP CSP students in this study were enrolled in the Code.org AP CSP course, which teaches JavaScript through App Lab, a dual-modality programming environment that allows learners to move back and forth between block-based and text-based presentations of the code.

3.3.3 AP Computer Science A. AP CS A is a year-long, traditional introduction to Java programming course that emphasizes features of the Java language as well as programming fundamentals. AP CS A is intended to be equivalent to a first-semester, college-level introductory computer science course and is often used as the summative course of a high school CS pathway.

3.3.4 Assessment Administration. Because these data are drawn from two different studies, the assessment administration procedure was not the same for all participants. The Intro to CS and AP CS A students completed the Mini-CA items online as part of a larger end-of-course computer science assessment that covered a wider range of constructs. The Mini-CA items were translated into Snap! and Java for inclusion in this larger assessment. The AP CSP students completed the entire Commutative Assessment online. It was embedded into their course using Code.org's content management system and teachers decided whether to assign the assessment to their students. Questions for students in the AP CSP condition were presented using the AP CSP block-based pseudocode (Figure 1c) and text-based pseudocode (Figure 1d).

4 FINDINGS

The findings section is broken down into two sections. First, we present statistics on assessment and item performance across languages and contexts, and then we present a deeper analysis of the differences we see by context. Table 2 shows student performance and Cronbach's alpha for the Mini-CA.

4.1 Summary Statistics

4.1.1 Overall assessment performance. Looking at the overall scores, a few things stand out. First, the Mini-CA was not difficult for the high school students in any of the three contexts, with mean scores ranging between 70.4% for AP CS A students to 83.4% for AP CSP students. Second, Cronbach's alpha remained high for all contexts. Cronbach's alpha is a measure of an assessment's reliability, or internal consistency. Values close to one indicate that the assessment items measure closely related constructs. The calculation for Cronbach's alpha is both item-dependent and sample-dependent,

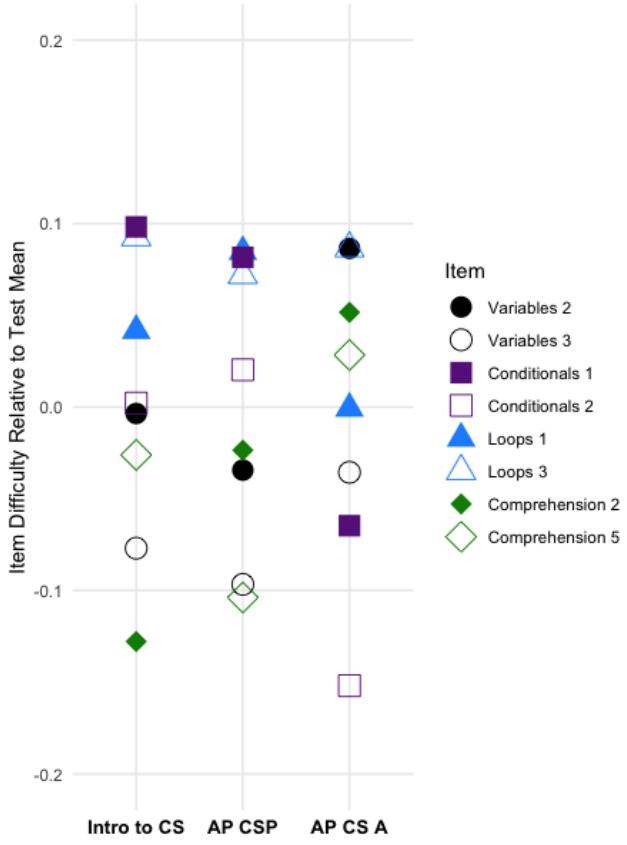


Figure 2: Normalized Mini-CA item difficulties by course

meaning that we cannot directly compare the values from course to course because the participants are different (i.e., we cannot say that the Mini-CA was "most reliable" for Intro to CS because it has the highest alpha coefficient). However, the high values for alpha across the board suggest the translation process did not significantly diminish the Mini-CA's reliability. Guidelines for interpreting Cronbach's alpha suggest that a value of 0.70 is of "modest" reliability and appropriate for early-stage research and a value of 0.80 is adequate for basic research studies [18, 21]. The alpha coefficients we report here are all within this range, even with the small number of items in the Mini-CA (Cronbach's alpha tends to be higher when there are more items in an assessment). This result suggests it is possible to translate an assessment into different languages without making it less reliable overall.

4.1.2 Relative item difficulty by course. Figure 2 shows the relative difficulty of each item in each context. These values were normalized by taking the total percent of students answering correctly on each item and then subtracting the mean percent correct for the entire Mini-CA. Thus, a value of zero on the y-axis indicates the item was of average difficulty for students in that course. Positive values indicate the item was easier than average and negative values indicate the item was more difficult than average. It should be noted that percent correct values and normed difficulties confound

Table 3: Distribution of incorrect responses for Variables 2.

#	Incorrect response	Intro to CS	AP CSP	AP CS A
1	x is equal to 15 y is equal to 15	35.4%	23.9%	45.5%
2	x is equal to 5 y is equal to 10	2.0%	21.8%	13.6%
3	x is equal to "x + 5" y is equal to "x"	35.4%	11.5%	9.1%
4	x is equal to 10, 15 y is equal to 5, 10	12.5%	16.5%	22.7%
5	x is equal to 10 y is equal to 5	14.6%	26.2%	9.1%

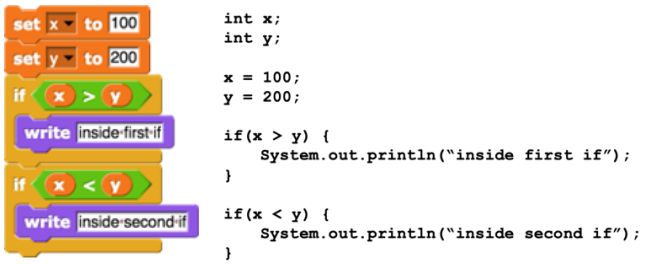
item difficulty and group proficiency. That is, a high percent correct value can indicate an easy item, a proficient group, or both. The three groups of students are not necessarily comparable in terms of their CS proficiency, so differences in item performance reflects both differences in the difficulty of the item in different languages as well as differences in group proficiency. Nevertheless, different rank orderings of items provides some indication that the relative difficulties of items change when they are translated into various languages.

For example, difficulty on item Comprehension 2 ranges from being very difficult for Intro to CS students to being relatively easy for AP CS A students. This may be an artifact of the respective curricula. Item Comprehension 2 asks students to read a function and explain its behavior; functions were likely not covered in-depth in the Intro to CS course while more time is spent on the topic in AP CS A. The fact that difficulty reflects content coverage suggests an initial sense of face validity for the item as it is administered in different context. We now take a deeper dive into some individual items, specifically looking at differences in performance by course and language. Due to space limitations, we focus on two of the items from the Mini-CA: Variables 2 and Conditionals 1.

4.2 Differences in Performance by Language

Variables 2 (shown in Figure 1) asks students to identify the values of x and y after the program executes. Variables 2 was of average difficulty or slightly more difficult than average for Intro to CS and AP CSP students, respectively, but it was the easiest item for AP CS A students. Table 3 shows the distribution of incorrect answers for this question (non-responders are omitted from the table). The error patterns differed noticeably between courses.

The third incorrect response option, x is equal to "x + 5" and y is equal to "x", was tied for the most common error made by students in Intro to CS, but tied for the least common error made by students in AP CS A. A potential explanation is that this is because Snap!, the language used in Intro to CS is a weakly-typed language and students do not necessarily recognize the difference between strings and variable expressions. Meanwhile Java, the language used for AP CS A is a strongly-typed language and the Java version of the question declares x and y as ints. Thus, AP CS A students who have a basic grasp of variable types would recognize that the third response option, which reports the values of x and y as strings, cannot be correct. This serves as one example of how the



(a) Snap! **(b) Java**

Figure 3: The code snippet for item Conditionals 1, presented in (a) Snap! and (b) Java.

Table 4: Distribution of incorrect responses for Cond'ls 1.

#	Incorrect response	Intro to CS	AP CSP	AP CS A
1	"inside first if"	21.4%	36.2%	57.7%
2	"inside first if" "inside second if"	57.1%	49.6%	34.6%
3	It will be different each time	21.4%	14.2%	7.7%

language of instruction can interact with conceptual knowledge and influence performance on an assessment, even when the main concept exists in both languages.

Figure 3 shows two of the four versions of item Conditionals 1. In this question, two variables, x and y, are initialized with values of 100 and 200, respectively. Students then have to trace the execution of code through two if statements and identify what the program will output. Table 4 shows the distribution of answers for students who responded to this item incorrectly.

The most common incorrect response for Intro to CS students and AP CSP students was the second option, which prints the contents inside both if statements. AP CS A students more frequently chose the first incorrect response, which prints the contents of the first if statement only (the correct answer is to print the contents of the second if statement only). We can only speculate about the reasons for this difference in error patterns, but it is worth mentioning that the Java version of the question is more difficult to parse visually than the other versions. The Java version uses the function "System.out.println" to generate output. This is less human-readable than the "write" block in the Snap! version. The AP CSP version uses a block/function called "display". Students viewing the Java version of the item also have to interpret the meaning of curly brackets to define scope, as opposed to having commands more clearly nested via the block-based code's graphical cues. This builds on earlier work on the Commutative Assessment that found students answering questions in the block-based modality performed better on questions related to conditional logic compared to students answering the same question in a text-based form [33].

4.3 Limitations

This analysis was conducted using available data and thus has some limitations. First, samples for each course differ in terms of

demographics and prior CS experiences. In addition, the Mini-CA is relatively easy and not likely to capture the full range of student learning in each course. This data set also does not allow us to tease apart the effects of programming language and curriculum, as only one course is included for each language. And finally, the sample sizes for Intro to CS and AP CS A were too small to use item response theory (IRT) to estimate item parameters such as difficulty [26]. IRT models are thought to be less sample-dependent than Cronbach's alpha. The analysis presented in this paper should not be used to draw conclusions about students' computer science proficiency, within a course or by comparing across courses, instead we present it to prompt debate. Different programming languages have different affordances and this influences students' responses to conceptually isomorphic questions. This understanding should inform how we design vertically-scaled assessments in computer science.

5 DISCUSSION

To more fully explore these findings and their implications for creating assessments that can serve to assess students learning at multiple points along a multi-year K-12 CS pathway, we begin our discussion with one of the categories of evidence for establishing an instrument's reliability and validity laid out by *Standards for Educational and Psychological Testing* [4], **evidence based on response process**. To establish an instrument's validity, assessment developers should have evidence that the strategies test takers use to provide their answers are relevant to the knowledge and skills the tasks are intended to measure. It seems likely that students responding to different-language versions of programming questions recruit slightly different processes to answer the questions. This is due to the fact that shifting from a text-based to a block-based modality introduces additional information to the question (e.g. shapes of blocks and how they fit together convey additional information around to meaning and behavior). Even if the prompts and code snippets are functionally equivalent across languages, differences in language syntax and semantics can lead students to recruit different knowledge and cognitive processes when answering the questions. Thus, it becomes difficult to disentangle a learner's knowledge about conditional logic from their general knowledge about how to parse block-based programs. These differences in response processes mean that it is likely not valid to treat different-language versions of the same item as if they were equivalent when establishing vertical scales.

5.1 Applying evidence-centered design

So where do we go from here? We continue our discussion with some ideas based on the evidence-centered assessment design (ECD) approach, which views assessment as an exercise in evidentiary reasoning. What student behaviors would reveal different levels of proficiency with the knowledge and skills being assessed [19]? ECD builds its Conceptual Assessment Framework around a series of models that define what is being measured and how. In this discussion we will focus on the student model, the evidence model, the task models, and the assembly model.

5.1.1 The student model. The student model describes what the assessment is intended to measure. For example, a simple student model might include a single variable, proficiency in a single domain, as characterized by the proportion of items the student is likely to answer correctly [19]. When developing vertically-scaled CS assessments, it may be necessary to create more complex student models that address a range of knowledge and skills in CS. For example, the student model could include variables for understanding of broader CS constructs (e.g., control structures) as well as concepts and skills that are more language-specific.

5.1.2 The evidence model. The evidence model describes how observable student responses to assessment tasks relate to their proficiency with the variables defined in the student model [19]. For example, the distractor options for a multiple-choice question could correspond to common misconceptions that would indicate lower proficiency with the related variables in the student model.

5.1.3 The task models. A task model describes how an assessment task should be structured in order to collect the evidence for the evidence model, including what material should be presented to the student as a part of the task prompt and what type of observable response the student should generate. Task models are developed for families of assessment tasks, not individual items. A single assessment can have multiple task models [19].

If a vertically-scaled CS assessment uses a complex student model, it seems to follow that the assessment would require a large number of task models in order to capture all the evidence needed for each variable in the student model, and/or include more complex tasks models. Currently, the most widely-used CS assessments tend to use a small number of task models, typically a code-reading task during which the student selects a single response from multiple options. A multi-lingual vertically-scaled assessment could include task models for non-coding items that assess foundational CS constructs as well as coding items. Differences in how students respond to these different tasks can then be used as evidence of proficiency for more general CS as well as language-specific concepts.

5.1.4 The assembly model. The assembly model puts it all together, describing how the other models "work together to form the psychometric backbone of the assessment" [19, p. 11]. The assembly model would describe the number of each item type (i.e., non-coding items vs. coding items in different languages) to ensure the assessment reflects the desired breadth and depth of the knowledge and skills in the student model while keeping the number of items to a manageable number per test administration.

5.2 Conclusion

We are not the first authors to take the position that CS education should develop vertically-scaled assessments that can measure student learning over time [31]. In this paper, we take that position farther and argue that such assessments must necessarily consider how students learn and transfer knowledge from one programming language to another. We discuss how the evidence-centered design approach could be used to guide the development of a multi-lingual, vertically-scaled assessment through the creation of more complex student models than we typically see in CS assessments and the use of more task models. Creating such an assessment presents a significant challenge to the field, but its existence would facilitate more rigorous, longitudinal evaluations of computer science pathways.

ACKNOWLEDGMENTS

This material is based in part upon work supported by the by the National Science Foundation under Grant #1348866. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] John R. Anderson. 1993. *Rules of the Mind.* Lawrence Erlbaum Associates, Hillsdale, NJ.

[2] Michal Armoni, Orni Meerbaum-Salant, and Mordechai Ben-Ari. 2015. From Scratch to 'Real' Programming. *ACM Transactions on Computing Education* 14, 4, Article 25 (2015), 15 pages. https://doi.org/10.1145/2677087

[3] Michal Armoni, Orni Meerbaum-Salant, and Mordechai Ben-Ari. 2015. From Scratch to "Real" Programming. *ACM Trans. Comput. Educ.* 14, 4, Article 25 (Feb. 2015), 15 pages. https://doi.org/10.1145/2677087

[4] American Educational Research Association, American Psychological Association, and National Council on Measurement in Education. 2014. *Standards for Educational and Psychological Testing.* AERA, Washington, DC.

[5] Guanhua Chen, Ji Shen, Lauren Barth-Cohen, Shiyan Jiang, Xiaoting Huang, and Moataz Eltoukhy. 2017. Assessing elementary students' computational thinking in everyday reasoning and robotics programming. *Computers & Education* 109 (2017), 162–175.

[6] D. C. Cliburn. 2008. Student opinions of Alice in CS1. In *2008 38th Annual Frontiers in Education Conference.* IEEE, T3B–1–T3B–6. https://doi.org/10.1109/FIE.2008.4720254

[7] Smarter Balanced Assessment Consortium. 2019. *Interpretive Guide for English Language Arts/Literacy and Mathematics Assessments.* https://portal.smarterbalanced.org/library/en/reporting-system-interpretive-guide.pdf

[8] Holger Danielsiek, Wolfgang Paul, and Jan Vahrenhold. 2012. Detecting and Understanding Students' Misconceptions Related to Algorithms and Data Structures. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education* (Raleigh, North Carolina, USA) *(SIGCSE '12).* Association for Computing Machinery, New York, NY, USA, 21–26. https://doi.org/10.1145/2157136.2157148

[9] Wanda Dann, Dennis Cosgrove, Don Slater, Dave Culyba, and Steve Cooper. 2012. Mediated transfer: Alice 3 to java. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education.* 141–146.

[10] Adrienne Decker and Monica M. McGill. 2019. A Topical Review of Evaluation Instruments for Computing Education. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (Minneapolis, MN, USA) *(SIGCSE '19).* Association for Computing Machinery, New York, NY, USA, 558–564. https://doi.org/10.1145/3287324.3287393

[11] Allison Elliott Tew. 2010. *Assessing fundamental introductory computing concept knowledge in a language independent manner.* Ph.D. Dissertation. Atlanta, GA. http://hdl.handle.net/1853/37090

[12] Allison Elliott Tew and Mark Guzdial. 2011. The FCS1: A Language Independent Assessment of CS1 Knowledge. In *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education* (Dallas, TX, USA) *(SIGCSE '11).* Association for Computing Machinery, New York, NY, USA, 111–116. https://doi.org/10.1145/1953163.1953200

[13] Mohammed F. Farghally, Kyu Han Koh, Jeremy V. Ernst, and Clifford A. Shaffer. 2017. Towards a Concept Inventory for Algorithm Analysis Topics. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education* (Seattle, Washington, USA) *(SIGCSE '17).* Association for Computing Machinery, New York, NY, USA, 207–212. https://doi.org/10.1145/3017680.3017756

[14] Diana Franklin, David Weintrop, Jennifer Palmer, Merijke Coenraad, Melissa Cobian, Kristan Beck, Andrew Rasmussen, Sue Krause, Max White, Marco Anaya, et al. 2020. Scratch Encore: The Design and Pilot of a Culturally-Relevant Intermediate Scratch Curriculum. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education.* 794–800.

[15] Dan Garcia, Brian Harvey, and Tiffany Barnes. 2015. The beauty and joy of computing. *ACM Inroads* 6, 4 (2015), 71–79.

[16] Mark Guzdial. 2019. *We Should Stop Saying 'Language Independent.' We Don't Know How To Do That.* https://cacm.acm.org/blogs/blog-cacm/238782-we-should-stop-saying-language-independent-we-dont-know-how-to-do-that/fulltext

[17] Brian Harvey and Jens Mönig. 2010. Bringing "no ceiling" to scratch: Can one language serve kids and computer scientists. *Proc. Constructionism* (2010), 1–10.

[18] Charles E. Lance, Marcus M. Butts, and Lawrence C. Michels. 2006. The Sources of Four Commonly Reported Cutoff Criteria: What Did They Really Say? *Organizational Research Methods* 9, 2 (2006), 202–220. https://doi.org/10.1177/1094428105284919

[19] Robert J. Mislevy, Russell G. Almond, and Janice F. Lukas. 2003. *A brief introduction to evidence-centered design.* Technical Report RR-03-16. Educational Testing Service, Princeton, NJ. https://www.ets.org/Media/Research/pdf/RR-03-16.pdf

[20] Mark Noone and Aidan Mooney. 2018. Visual and textual programming languages: a systematic review of the literature. *Journal of Computers in Education* 5 (2018), 149–174. https://doi.org/10.1007/s40692-018-0101-5

[21] Jum Nunnally and Ira H. Bernstein. 1994. *Psychometric Theory: 3rd Edition.* McGraw-Hill Education.

[22] Miranda C. Parker, Mark Guzdial, and Shelly Engleman. 2016. Replication, Validation, and Use of a Language Independent CS1 Knowledge Assessment. In *Proceedings of the 2016 ACM Conference on International Computing Education Research* (Melbourne, VIC, Australia) *(ICER '16).* Association for Computing Machinery, New York, NY, USA, 93–101. https://doi.org/10.1145/2960310.2960316

[23] Miranda C. Parker, Yvonne S. Kao, Dana Saito-Stehberger, Diana Franklin, Susan Krause, Debra Richardson, and Mark Warschauer. 2021. Development and Preliminary Validation of the Assessment of Computing for Elementary Students (ACES). In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education* (Virtual Event, USA) *(SIGCSE '21).* Association for Computing Machinery, New York, NY, USA, 10–16. https://doi.org/10.1145/3408877.3432376

[24] Leo Porter, Daniel Zingaro, Soohyun Nam Liao, Cynthia Taylor, Kevin C. Webb, Cynthia Lee, and Michael Clancy. 2019. BDSI: A Validated Concept Inventory for Basic Data Structures. In *Proceedings of the 2019 ACM Conference on International Computing Education Research* (Toronto ON, Canada) *(ICER '19).* Association for Computing Machinery, New York, NY, USA, 111–119. https://doi.org/10.1145/3291279.3339404

[25] Arif Rachmatullah, Bita Akram, Danielle Boulden, Bradford Mott, Kristy Boyer, James Lester, and Eric Wiebe. 2020. Development and validation of the middle grades computer science concept inventory (MG-CSCI) assessment. *Eurasia Journal of Mathematics, Science and Technology Education* 16, 5 (2020). https://doi.org/10.29333/ejmste/116600

[26] Alper Sahin and Duygu Anil. 2017. The effects of test length and sample size on item parameters in item response theory. *Educational Sciences: Theory & Practice* 17, 1 (2017). https://doi.org/10.12738/estp.2017.1.0270

[27] Elizabeth Schofield, Michael Erlinger, and Zachary Dodds. 2014. MyCS: CS for Middle-Years Students and Their Teachers. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education* (Atlanta, Georgia, USA) *(SIGCSE '14).* Association for Computing Machinery, New York, NY, USA, 337–342. https://doi.org/10.1145/2538862.2538901

[28] Jean Scholtz and Susan Wiedenbeck. 1990. Learning second and subsequent programming languages: A problem of transfer. *International Journal of Human-Computer Interaction* 2, 1 (1990), 51–72.

[29] Exploring Computer Science. [n.d.]. *What is ECS?* http://www.exploringcs.org/

[30] Juha Sorva. 2012. *Visual program simulation in introductory programming education; Visuaalinen ohjelmasimulaatio ohjelmoinnin alkeisopetuksessa.* G4 Monografiaväitöskirja. http://urn.fi/URN:ISBN:978-952-60-4626-6

[31] Rebecca Vivian, Diana Franklin, Dave Frye, Alan Peterfreund, Jason Ravitz, Florence Sullivan, Melissa Zeitz, and Monica M. McGill. 2020. Evaluation and Assessment Needs of Computing Education in Primary Grades. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education* (Trondheim, Norway) *(ITiCSE '20).* Association for Computing Machinery, New York, NY, USA, 124–130. https://doi.org/10.1145/3341525.3387371

[32] David Weintrop, Heather Killen, and Baker E Franke. 2018. Blocks or Text? How programming language modality makes a difference in assessing underrepresented populations. In *Rethinking Learning in the Digital Age: Making the Learning Sciences Count, 13th International Conference of the Learning Sciences (ICLS) 2018.* International Society of the Learning Sciences. https://doi.org/10.22318/cscl2018.328

[33] David Weintrop and Uri Wilensky. 2015. Using Commutative Assessments to Compare Conceptual Understanding in Blocks-Based and Text-Based Programs. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research* (Omaha, Nebraska, USA) *(ICER '15).* Association for Computing Machinery, New York, NY, USA, 101–110. https://doi.org/10.1145/2787622.2787721

[34] David Weintrop and Uri Wilensky. 2019. Transitioning from introductory block-based and text-based environments to professional programming languages in high school computer science classrooms. *Computers & Education* 142 (2019), 103646. https://doi.org/10.1016/j.compedu.2019.103646

[35] Linda Werner, Jill Denner, Shannon Campe, and Damon Chizuru Kawamoto. 2012. The Fairy Performance Assessment: Measuring Computational Thinking in Middle School. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education* (Raleigh, North Carolina, USA) *(SIGCSE '12).* Association for Computing Machinery, New York, NY, USA, 215–220. https://doi.org/10.1145/2157136.2157200

[36] Eric Wiebe, Jennifer London, Osman Aksit, Bradford W. Mott, Kristy Elizabeth Boyer, and James C. Lester. 2019. Development of a Lean Computational Thinking Abilities Assessment for Middle Grades Students. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (Minneapolis, MN, USA) *(SIGCSE '19).* Association for Computing Machinery, New York, NY, USA, 456–461. https://doi.org/10.1145/3287324.3287390