Block-based Comprehension: Exploring and Explaining Student Outcomes from a Read-only Block-based Exam

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

Heather Killen
University of Maryland
College Park, MD, USA
hkillen@umd.edu

Talal Munzar
University of Maryland College Park, MD, USA
tmunzar@terpmail.umd.edu

Baker Franke
Code.org
Seattle, WA, USA
baker@code.org

ABSTRACT

The success of block-based programming environments like Scratch and Alice has resulted in a growing presence of the block-based modality in classrooms. For example, in the United States, a new, nationally-administered computer science exam is evaluating students' understanding of programming concepts using both block-based and text-based presentations of short programs written in a custom pseudocode. The presence of the block-based modality on a written exam in an unimplemented pseudocode is a far cry from the informal, creative, and live coding contexts where block-based programming initially gained popularity. Further, the design of the block-based pseudocode used on the exam includes few of the features cited in the research as contributing to positive learner experiences. In this paper, we seek to understand the implications of the inclusion of an unimplemented block-based pseudocode on a written exam. To do so, we analyze responses from over 5,000 students to a 20 item assessment that included both block-based and text-based questions written in the same pseudocode as the national exam. Our analysis shows students performing better on questions presented in the block-based form compared to text-based questions. Further analysis shows that this difference is consistent across conceptual categories. This paper contributes to our understanding of the affordances of block-based programming and if and how the modality can help learners succeed in early computer science learning experiences.

ACM Reference format:

D. Weintrop, H. Killen, T. Munzar, and B. Franke. 2019. Block-based Comprehension: Exploring and Explaining Student Outcomes from a Read-only Block-based Exam In Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE '19). *February 27-March 2, 2019, Minneapolis, MN, USA,* ACM, NY, NY, USA, 7 pages. https://doi.org/10.1145/3287324.3287348

KEYWORDS

Introductory Programming Environments; High School Computer Science Education; Block-based Programming; Assessment

1 INTRODUCTION

The last five years have seen a steady flow of block-based programming tools into K-12 classrooms. While some block-based programming environments have a long history in formal educational contexts (e.g. Alice), other block-based tools were specifically designed for informal learning spaces (e.g. Scratch). As part of the transition of block-based programming into K-12 classrooms, the modality is starting to be used in ways quite distinct from how it was initially designed. Nowhere is this clearer than when it comes to assessment.

Many introductory computer science courses assess student knowledge through written exams that ask students questions about specific syntactic features of a programming language and evaluate student comprehension of programs. While not ideal, such questions lend themselves well to the multiple-choice question format and thus can be graded quickly and objectively. As a result, written, multiple choice assessments are common in introductory computing contexts.

The rise of block-based programming environments in classrooms presents an interesting challenge for educators. What happens when we use learning environments designed for informal spaces that prioritize creativity and expression and situate them in formal contexts where they are subject to conventional educational institutional constraints, such as summative written examinations? Often the result is for educators to create pen-and-paper written exams or other static presentations of material based on the block-based programming environments students used in their classrooms. In moving from the programming environment itself to the written assessment context, many of the affordances of these environments are lost as the graphical representation is recreated in static, often black-and-white printed exam booklets. This raises interesting research questions on the affordances of block-based tools and if and how they support learning beyond the programming environment itself. It is these questions we are pursuing with this research. Specifically, we seek to answer the following questions:

> *Does the block-based programming modality support novice programmers' program comprehension on static, read-only assessments? And, if so, how?*

To answer these questions, we designed a 20-item assessment asking questions using both block-based and text-based forms of a custom pseudocode (Fig. 1). The exam was then included as an optional activity in materials distributed nationally, resulting in over 5,000 students completing the assessment.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
SIGCSE '19, February 27-March 2, 2019, Minneapolis, MN, USA
© 2019 Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-5890-3/19/02…$15.00
https://doi.org/10.1145/3287324.3287348

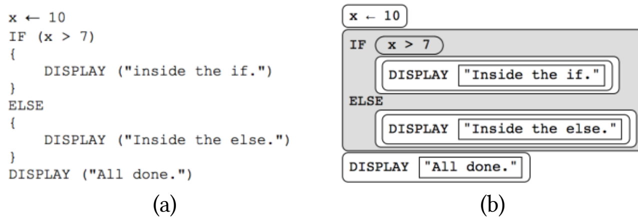

Figure 1. A sample text-based (left) and block-based (right) program written in the AP CSP pseudocode.

The paper begins with a review of relevant literature. We then describe the block-based pseudocode that was used in the assessment and present a comparison of the pseudocode to widely-used block-based languages to understand how they are similar and where they diverge. Next, we present information on the assessment we created and the methods and participants of the study. The findings section follows and the paper concludes with potential explanations for the findings and a discussion of the implications of this work.

2. PRIOR WORK

2.1 The Role of Representation in Learning

The first and most foundational literature that this paper builds on is prior work studying the role of representation in learning. This work has come under a few labels. Kaput and colleagues use the term *representational system* in their investigation of the cognitive impacts of different symbol systems used in mathematics and how they affect cognitive aspects of engaging with mathematics [21]. Moving beyond the individual, they expand this idea through the concept of *representational infrastructure* as a way to talk about how a representational system supports the cultural and social dimensions of thinking and communicating about ideas [20]. Much of this work is specifically in relation to technology and the new representations and interaction patterns the medium affords. In his conceptualization of *Computational Literacy*, diSessa builds on this idea with the notion of material intelligence, saying "we don't always have ideas and then express them in the medium. We have ideas *with* the medium" [7, emphasis in the original].

While often assumed to be static, Wilensky and Papert show how representational infrastructures can and should change over time [46]. As part of their *Restructuration Theory*, they present examples of such representational shifts, or restructurations, and provide criteria to evaluate the different capacities that representational systems play. Our exploration of the impact of presenting programs using block-based representations builds directly on this work as it is trying to make sense of representational affordances and understand if and how the way ideas are represented shapes learners ability to interpret the ideas presented. In doing so, this work shares a goal with Sherin's investigation into the role of representations in learning physics [33] and Gilmore and Green in looking at declarative versus procedural notation [13].

2.2 Block-based Programming

The second body this study builds off of and contributes to is work on block-based programming. Block-based programming (Fig. 2) is a graphical programming modality that presents programming commands as visual blocks that can be assembled via a drag-and-drop interaction [3]. The environment provides a direct manipulation interface for authoring programs [34]. While not a recent innovation (e.g. [4]), the block-based approach to programming has become widespread in introductory contexts due to the success of tools such as Scratch [30], Alice [6], and the Blockly library [12].

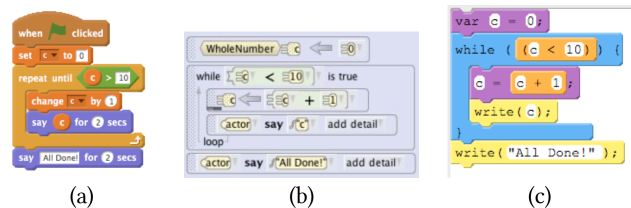

Figure 2. Three examples of block-based programming environments: (a) Scratch (b) Alice and (c) Code.org's AppLab.

A growing body of research is revealing how, where, and in what ways block-based programming is effective for introducing novices to the practice of programming and the field of computer science more broadly. This includes research documenting novice programmers' learning gains [10, 17, 29, 42], as well as attitudinal and perceptual shifts in interest in the field of computer science [22, 25, 26]. Research has also investigated how block-based environments can serve as a context around which participatory cultures can form, further bolstering learners identities as computational doers [9, 19, 32]. Section 4 of this paper presents a more detailed account of the specific affordances of block-based programming as these design features are central to this paper.

2.3 Evaluating Programming Knowledge

In their review of assessments of introductory programming, Gross and Powers [14] found that "some of the least studied questions are those that focus on how the environments impact a student's learning process and understanding from a formative perspective." In the decade since that statement was written, a growing body of research has started to answer some of these open questions [16]. This work includes creating and validating traditional summative assessments that use multiple choice questions to assess knowledge (e.g. [26]), performance-based assessments where students are asked to complete a specific task (e.g. [45]), and portfolio or artifact-based assessments that used student-produced work as a way to measure students level of understanding (e.g. [5]).

The work presented in this paper most closely builds off research into program comprehension of statically presented programs used in summative assessments. This includes assessments such as the Foundations for Advancing Computational Thinking assessment [17] which has been used to

investigate learner misconceptions in block-based programs [15]. A second line of work this study builds off is research comparing block-based and text-based comprehension using the Commutative Assessment [44]. Studies using the Commutative Assessment have shown how students perform better on questions presented in block-based forms, even when students' preceding programming instruction was in a text-based programming environment [42]. The work presented in this paper contributes to this literature by focusing on a relatively feature-light block-based programming language: the AP Computers Science Principles Pseudocode.

3. THE AP COMPUTER SCIENCE PRINCIPLES PSEUDOCODE

In the 2016-2017 school year, the College Board introduced a new Advanced Placement (AP) course to be taught in the United States called Computer Science Principles (CSP) [2]. The course focuses on seven big ideas of computer science, meaning, unlike many other introductory computing courses, AP CSP does not prioritize programming over other computer science content. One feature of the course is that it is programming language agnostic, meaning teachers are free to choose the programming language and environments they will use in instruction. This presents a challenge for the nationally-administered summative assessment of the course as the programming questions need to reflect the programming plurality welcomed in the design of the curriculum.

In response to this challenge, the AP CSP development committee invented a pseudocode that had both text-based and visual block-based representations (Fig. 1). The summative AP CSP exam includes questions asked in BOTH block-based and text-based modalities, meaning all students answer questions asked in both forms. The goal of this decision is to ensure that students are not rewarded or penalized for using one type of programming tool or another during the school year.

The AP pseudocode developed for the AP CSP exam consists of 23 keywords including looping constructs (e.g. REPEAT), conditional operators (e.g. IF), and I/O (e.g. DISPLAY). The block-based and text-based representations of the commands are isomorphic, meaning that anything that can be represented in one form can also be represented in the other. The place the two representations differ with respect to the symbols used is in relation to scope, where the text form uses ()s, []s, and {}s while the block-based form conveys this information through the boundaries of the blocks. In the opinion of the authors, the block-based form of the pseudocode can be described as the text-based form with ovals and rectangles drawn around the commands. The full documentation for this pseudocode can be found on pages 14-20 of the AP Computer Science Principles Assessment Overview and Performance Task Directions for Students [1].

4. AFFORDANCES OF BLOCK-BASED PROGRAMMING

In this section, we review specific features of block-based tools that the literature has documented as supporting novices and discuss if and how it is present or absent in the AP CSP pseudocode. Following the strategy of Robins et al. in their review of the literature and learning and teaching programming [31], we break the affordances into two main categories: supports for program comprehension and supports for program generation, although we recognize some design features may support both roles. The following list of features is compiled from a number of resources delineating the characteristics and affordances of block-based environments [3, 8, 11, 24, 27, 37, 43]. After discussing the design feature, we briefly discuss if and how the feature is present in the AP CSP pseudocode that is the focus of this study.

4.1 Supports for Program Comprehension

4.1.1 Visual Rendering of Blocks. The defining feature of block-based programming environments is that each command is rendered as a block. These blocks present visual cues denoting information about the command. This information includes a specific shape showing how and where the block can be used (e.g. commands having notches at the top and bottom, Boolean variables presented as hexagons, and "hat" blocks having rounded tops to make clear that nothing can precede it). Similarly, blocks that accept arguments can render their argument slots using the same shape, making it clear the types of inputs the block will accept. Blocks are also often color-coordinated, so families of blocks related to a concept share a color (e.g. all control blocks in Scratch are yellow). Block-based environments also use shape to denote scope through "c-shaped" blocks that wrap commands and render the nested inside the structure as can be seen in Fig. 2a and Fig. 2c.

While the AP CSP block-based pseudocode includes some of these features, such as encircling commands with lines to give the appearance of blocks and nesting blocks inside each other, most of the visual features cited in the literature are absent. For example, the AP CSP blocks do not include notches to denote how blocks fit together and do not use color to convey the block category.

4.1.2 Natural Language Block Labels. Another oft-cited affordance of block-based environments is their ability to use natural language expressions within the command themselves. Since the block rendering itself defines the scope of each command and how it is to be parsed by the compiler, the language designer is free of the syntactic and keyword constraints of text-based languages. As a result, it is possible to create block-based commands that read like sentences (e.g. the `change x by 1` block shown in Fig. 2a) as well as domain-specific block languages with commands tailored to the specific domain (e.g. Frog Pond [18] or CoBlox [39]).

Given the constraint that the AP CSP pseudocode needed to support both block-based and text-based forms, the pseudocode was not able to take advantage of this feature of the block-based modality. This can be seen in the character-by-character similarities between the programs shown in Fig. 1a and Fig. 1b.

4.1.3 Browsability. The third feature of block-based tools we highlight is not a characteristic of the language itself, but rather, a feature of the larger block-based apparatus that makes block-based programming possible. Block-based programming environments present users with the set of available blocks in a logically ordered and easily browsed set of "drawers" since blocks need to be

present somewhere on the screen in order for the user to drag them into their program. One benefit of this feature is that it makes blocks "browsable", meaning learners need not have commands memorized, but instead can easily discover blocks, which can then be used to bootstrap programming ideas.

A version of this feature is available to students for the AP CSP pseudocode. During the exam, students have access to a reference sheet describing the pseudocode and the keywords that comprise the language, meaning they can "browse" the blocks. However, the static, black-and-white reference sheet is a simplified version of the interface of most block-based environments.

4.2 Supports for Program Generation

In this section, we document additional features of block-based programming that the literature has identified as contributing to making it an effective way to introduce novices to programming. Due to the fact that the AP CSP pseudocode was only designed for comprehension on a written test, the language itself has never been implemented. Thus, none of the benefits listed in this section are present in the AP CSP Pseudocode.

4.2.1 Drag-and-Drop Composition. Unlike text-based languages where statements are typed in one character at a time, authoring programs in block-based tools allows learners to assemble commands using a drag-and-drop interaction. The information provided by the visual rendering of the command gives the user information about how and where a block can be used. If two blocks cannot be joined to form a valid statement, then the block-based editor prevents the commands from snapping together, thus preventing most types of syntax errors. This drag-and-drop construction also lends a feeling of "tinkerability" and playfulness, as changes can be made quickly and easily, especially for novice computer users who may find typing commands cumbersome.

4.2.2 Dynamic Rendering. Given the graphical nature of the block-based representation, the visual presentation of commands allows for a number of additional dynamic rendering features that can support the user. For example, the shapes of blocks can change to fit changing characteristics of the program (e.g. c-shaped blocks can grow to wrap a larger number of sub-commands). Additionally, block-based tools can provide hover-over tooltips, include images or short animations, or allow the user to modify the shape of the block on the fly (e.g. Blockly's mutator feature). While such features are possible in text-based editors (e.g. Citrus [23]), they are not as widely used or as central to the modality as the dynamic rendering features of block-based tools.

5. METHODS AND PARTICIPANTS

To answer our stated research questions we gathered data on students answering questions in both the block-based and text-based forms of the AP CSP Pseudocode. To do so, we created a 20 question multiple choice assessment in the same form as the multiple choice question portion of the AP CSP written exam. We then included the assessment as an optional "Practice AP Programming exam questions" module at the end of the programming module of Code.org's CSP curriculum (http://code.org/csp). Code.org's CSP curriculum is a full-year, rigorous, entry-level course that introduces high school students to the foundations of modern computing and prepares them for the AP CSP Exam. Programming in the course is done in App Lab (Fig. 2c, http://code.org/applab), which is a JavaScript-based, dual-modality environment where students can construct programs in both block-based and text-based modalities.

5.1 The AP CSP Pseudocode Assessment

The assessment used in this study is comprised of 20 multiple choice questions, 10 in the block-based form and 10 in the text-based form of the AP CSP pseudocode. The questions were drawn from a previously used and validated assessment used in similar research studies [41]. Each question on the assessment begins with a short code snippet and is followed by the question: "What will the output of the program be?" There are also a series of questions that present short programs that use the prompt "What does this program do?" These comprehension questions are intended to evaluate a student's ability to identify the purpose of a program as opposed to just mentally running the program and reporting the output. The assessment covers five programming topics: variables, loops, conditionals, functions, and program comprehension. For each of the five topics, the assessment asked two questions in the block-based pseudocode form and two in the text-based pseudocode. This counter-balance design ensures that every student answered two questions for each concept in both forms of the pseudocode. A sample question is shown in Fig. 1.

5.2 Data collection and participants

The assessment was administered through Code.org's content management system that tracks individual students as well as classroom progress through the curriculum. At the beginning of the course, students create a profile which includes optionally self-reporting their gender, age, and race. The responses collected by Code.org were de-identified and then shared with researchers. All necessary institutional approval was acquired for conducting this research.

The dataset for this study consists of 5,427 students and over 105,000 individual question responses. The sample includes 1,218 (22.4%) female students, 3,198 (58.9%) male students, and 1,011 students (18.6%) who chose to not provide gender information. Of the 5,427 students, 1,040 (19.2%) learners self-identified as being from an underrepresented minority in computing (URM), while 2,199 (40.5%) of students were classified as not URM and 2,188 (40.3%) of student did not self-report their race. For this work, students that self-identified as Black, Hispanic, LatinX, Native American, or Pacific Islander were categorized as being from a URM. Finally, a majority of participants were between the ages of 15 and 18, which corresponds to the four years of American high-school (9.6% 15 years old; 22.5% 16 years old; 30.7% 17 years old; 27.2 % 18 years old).

6. FINDINGS

6.1 Performance by Modality

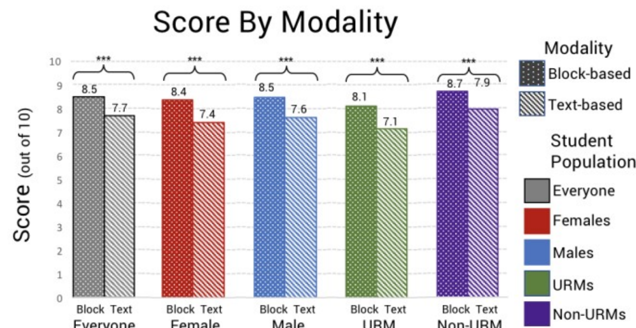

Figure 3. Average scores on the assessment by modality.

6.1.1 Overall Performance. To answer our first research question on the impact of modality, we first look at overall student performance on the 20-question assessment. The leftmost column of Fig. 3 shows student performance on the block-based questions compared to the text-based questions. Overall, students scored an average of 8.5 out of 10 on block-based questions (*SD* 1.9) and 7.7 out of 10 on text-based questions (*SD* 2.2), a difference that is statistically significant $t(4761) = 38.14$, $p < .001$, $d = .40$. This means, overall, students performed better on the block-based questions than the text-based questions.

6.1.2 Performance by Gender. Looking at gender, we find that both male and female students perform significantly better on block-based questions: Female $t(1081) = 20.25$, $p < .001$, $d = 0.62$ and Male $t(2852) = 29.344$, $p < .001$, $d = 0.55$ (columns 2 and 3 in Fig. 3 respectively). In the case of female students, this resulted in a .92 point improvement on average while males had a .85 point increase, as can be seen by the larger effect size for female students.

6.1.3 Performance by Race. When conducting the same analysis and dividing the participant population by race, we see a similar pattern. Both URM and non-URM students performed significantly better on block-based questions versus text-based questions: URM students $t(878) = 18.20$, $p < .001$, $d = 0.61$ and non-URM students $t(2000) = 23.74$, $p < .001$, $d = 0.53$ (the two right-most pairs of columns in Fig. 3). For participants that self-identified as a URM, there was, on average a full point difference between mean block-based score and mean text-based score. Non-URM students saw on average score improvement of .76 points between block-based and text-based questions. A longer, more detailed analysis of the impact of modality on learners from historically underrepresented populations can be found in [41].

6.1.4 Performance by Concept. Along with analyzing results by characteristics of the learner, we can also look at student performance by modality across the concept being assessed. Fig. 4 reports the percentage of students that got each question correct grouped by modality and concept.

Across all five content areas, students scored better on questions asked in the block-based modality over the text-based modality. Given students only answered 2 questions for each modality-concept pairing, we present only averages in Fig. 4. A redesign of the assessment is underway that will give us greater power to further investigate the interaction of modality and concept in code comprehension questions.

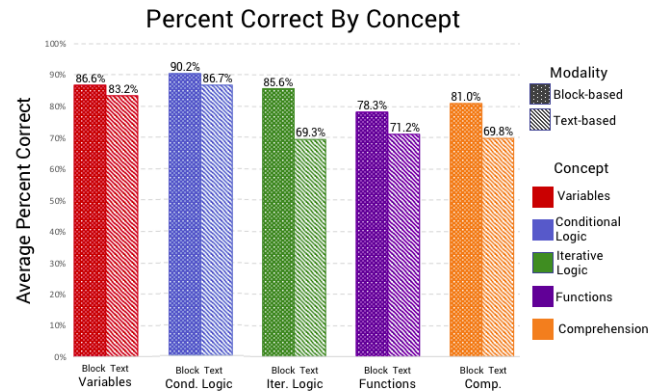

Figure 4. Average number of students who answered a question correctly grouped by concept and modality

Taken together, this collection of results is slightly surprising as the block-based format used in this assessment lacks many of the features the literature has identified as making block-based tools easier to comprehend. This suggests that the features that are present, namely the block-shapes created by outlining each command, play a central role in facilitating the comprehension of programs by novice programmers.

7. DISCUSSION

7.1 Potential Explanations for these Findings

The main finding from this study is that students do better on questions presented in a block-based form compared to isomorphic text-based presentations. This on its own is not that surprising as previous studies have found similar results (e.g. [44]). What is surprising is that we see these same patterns using the AP CSP pseudocode block-based presentation which includes relatively few of the features that learners and researchers cite as being responsible for the ease-of-comprehension. Further, the AP CSP pseudocode is not a real language, meaning learners have never written a program using it. In this section, we explore a number of potential explanations for the findings presented.

7.1.1 The Simple Visual Cues are Sufficient. The first potential explanation is that the basic visual cues that are present in the AP CSP pseudocode are enough to cause the differences found in the data. By visual cues, we are referring to both the rectangular and oval lines encircling commands that give them their block-based appearance and the nesting of commands. It is also possible that the opposite is true, that the absence of potentially confusing or difficult-to-parse language features that are present in the text-based questions, such as `{}`s and `()`s are the cause of student difficulty in the text-based form. This explanation is supported by

prior research on the difficulty novices have with syntactic features of programming languages, including the use of less familiar symbols like the brackets and braces [36].

7.1.2 The Block-based Pseudocode is Closer to Prior Programming Experiences. While the AP CSP Pseudocode used on the assessment was created specifically for the written test, students did have prior programming experience that could have shaped these results. For example, if students had spent the two months leading up to the exam programming in a block-based environment such as Scratch it is possible that would make them more likely to do better on the block-based questions. Similarly, if students came into the test on the heels of a unit learning JavaScript, the text-based questions may appear more familiar and thus easier. While the AP CSP exam was designed to prevent this, it is still possible.

The assessment used in this study was embedded in a curricular unit based on Code.org's AppLab environment (Fig. 2b). While AppLab includes both a block-based and text-based interface, research shows that novices generally prefer the block-based modality [28, 40, 43], which also matches the anecdotal evidence received from teachers of the course. Given these facts, the prior experience explanation is certainly plausible. However, prior research has not found a strong coupling of the modality used for learning programming and ability to answer questions on a static exam. In two separate studies, Weintrop and Wilensky found students who worked in block-based environments performed better on text-based programming questions than peers who had learned using text-based programming tools [42, 44].

7.1.3 Block-based Programs are Friendlier. A third potential explanation focuses not on a feature of the language itself or on learners' prior experiences, but on learners perception of the language and how they relate to it. This explanation draws from the literature on stereotype threat, which finds that students underperform when there is a risk of their performance confirming an existing stereotype [35]. This explanation posits that when learners see the block-based pseudocode, the form is closer to the fun and playful programming they relate to, as opposed the serious and professional programming languages they make them feel unwelcome. While there is more work that needs to be done to confirm (or refute) this account, it does provide a plausible explanation for the differences observed in the analysis of gender and race presented in sections 6.1.2 and 6.1.3.

7.1.4 Classroom or Curricular Factors. Given that this study took place in classrooms across the United States there are many other possible environmental factors that may contribute to these findings. For example, it is possible that teacher effects played a role in this pattern. If a teacher was new to computer science, he or she may have felt more comfortable with, and therefore chosen to work in the block-based modality, resulting in block-based examples during classroom instruction. Likewise, the curriculum used in the classroom may have prioritized the block-based modality over the text-based form.

While we think there is merit to all these explanations and suspect the truth includes a bit of all of them. More work remains to be done to further tease apart these potential explanations.

7.2 Broadening Participation in Computing

A second important discussion point from this work is how these findings speak to the goal of broadening participation in computing. One of the objectives of the AP CSP course was to introduce learners from populations historically underrepresented in computing to the field of computer science. By framing the course around seven-big ideas, as opposed to just programming, the hope was to present a broader (and more accurate) picture of what computer science is. From this perspective, the finding that students who self-identify as members of historically underrepresented groups, like female, Black and Hispanic learners, performed better on the block-based form is important. Having early successes in computing experiences, especially for AP classes, which are designed to be more challenging than traditional high school coursework, has the potential to positively impact their attitudes towards the field and reshape how they perceive themselves with respect to whether or not they are (or can be) a computer scientist.

8. IMPLICATIONS AND CONCLUSION

As this work was based on a newly introduced exam taken by thousands of learners across the United States, there are a number of implications for the findings. First, the finding that learners perform better on questions presented in the block-based modality has direct implications for educators and their choice of learning environment and curricula. Second, the findings have implications for designers of introductory programming tools and their decisions about whether to include block-based features and if so, how. This research suggests that even the most basic block-based features are enough to improve learning outcomes. A final potential implication for this work is related to assessing emerging computing knowledge. The results from this research suggest that the block-based pseudocode created for the exam was successful in providing supports for novices. This means efforts to include features of block-based tools into assessment contexts is a potentially productive decision.

The growing interest in computing education across the K-12 spectrum has produced a variety of initiatives intended to bring computer science to all. As part of this larger effort, educators, curriculum designers, and assessment creators have drawn inspiration from the successes and popularity of block-based tools as a way to introduce learners to computer science in formal contexts. This paper provides further evidence that the block-based modality does help learners on static written assessments. Further, it is particularly helpful for learners from historically underrepresented groups. This finding validates some of the decisions made by the designers of the AP CSP written exam. With this finding, we continue to advance our understanding of the role block-based programming might play in helping learners at the beginning of their journey into the world of computer science.

9. REFERENCES

[1] AP Computer Science Principles Assessment Overview and Performance Task Directions for Students: *https://apcentral.collegeboard.org/pdf/ap-csp-student-task-directions.pdf*. Accessed: 2018-04-06.

[2] Arpaci-Dusseau, A. et al. 2013. Computer Science Principles: Analysis of a Proposed Advanced Placement Course. *Proceeding of the 44th ACM Technical Symposium on Computer Science Education* (New York, NY, USA, 2013), 251–256.

[3] Bau, D., Gray, J., Kelleher, C., Sheldon, J. and Turbak, F. 2017. Learnable programming: blocks and beyond. *Communications of the ACM*. 60, 6 (May 2017), 72–80. DOI:https://doi.org/10.1145/3015455.

[4] Begel, A. 1996. *LogoBlocks: A graphical programming language for interacting with the world*. Electrical Engineering and Computer Science Department. MIT.

[5] Brennan, K. and Resnick, M. 2012. New frameworks for studying and assessing the development of computational thinking. (Vancouver, Canada, 2012).

[6] Cooper, S., Dann, W. and Pausch, R. 2000. Alice: a 3-D tool for introductory programming concepts. *Journal of Computing Sciences in Colleges*. 15, 5 (2000), 107–116.

[7] diSessa, A.A. 2000. *Changing minds: Computers, learning, and literacy*. MIT Press.

[8] Dwyer, H., Hill, C., Hansen, A., Iveland, A., Franklin, D. and Harlow, D. 2015. Fourth Grade Students Reading Block-Based Programs: Predictions, Visual Cues, and Affordances. *Proceedings of the eleventh annual International Conference on International Computing Education Research* (2015), 111–119.

[9] Fields, D., Giang, M. and Kafai, Y. 2014. Programming in the wild: trends in youth computational participation in the online Scratch community. *Proceedings of the 9th Workshop in Primary and Secondary Computing Education* (2014), 2–11.

[10] Franklin, D., Skifstad, G., Rolock, R., Mehrotra, I., Ding, V., Hansen, A., Weintrop, D. and Harlow, D. 2017. Using Upper-Elementary Student Performance to Understand Conceptual Sequencing in a Blocks-based Curriculum. *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education* (New York, NY, USA, 2017), 231–236.

[11] Fraser, N. 2013. *Blockly*. Google.

[12] Fraser, N. 2015. Ten things we've learned from Blockly. *2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)* (Oct. 2015), 49–50.

[13] Gilmore, D.J. and Green, T.R.G. 1984. Comprehension and recall of miniature programs. *International Journal of Man-Machine Studies*. 21, 1 (Jul. 1984), 31–48. DOI:https://doi.org/10.1016/S0020-7373(84)80037-1.

[14] Gross, P. and Powers, K. 2005. Evaluating Assessments of Novice Programming Environments. *Proceedings of the First International Workshop on Computing Education Research* (New York, NY, USA, 2005), 99–110.

[15] Grover, S. and Basu, S. 2017. Measuring Student Learning in Introductory Block-Based Programming: Examining Misconceptions of Loops, Variables, and Boolean Logic. *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education* (New York, NY, 2017), 267–272.

[16] Grover, S., Cooper, S. and Pea, R. 2014. Assessing computational learning in K-12. (2014), 57–62.

[17] Grover, S., Pea, R. and Cooper, S. 2015. Designing for deeper learning in a blended computer science course for middle school students. *Computer Science Education*. 25, 2 (Apr. 2015), 199–237. DOI:https://doi.org/10.1080/08993408.2015.1033142.

[18] Horn, M.S., Brady, C., Hjorth, A., Wagh, A. and Wilensky, U. 2014. Frog pond: a codefirst learning environment on evolution and natural selection. *Proceedings of the 2014 conference on Interaction design and children* (2014), 357–360.

[19] Kafai, Y.B. and Burke, Q. 2014. *Connected Code: Why Children Need to Learn Programming*. MIT Press.

[20] Kaput, J., Noss, R. and Hoyles, C. 2002. Developing new notations for a learnable mathematics in the computational era. *Handbook of international research in mathematics education*. (2002), 51–75.

[21] Kaput, J.J. 1987. Towards a Theory of Symbol. *Problems of Representation in the Teaching and Learning of Mathematics*. C. Janvier, ed. Lawrence Erlbaum Associates. 159.

[22] Kelleher, C., Pausch, R. and Kiesler, S. 2007. Storytelling alice motivates middle school girls to learn computer programming. *Proceedings of the SIGCHI conference on Human factors in computing systems* (2007), 1455–1464.

[23] Ko, A.J. and Myers, B.A. 2005. Citrus: a language and toolkit for simplifying the creation of structured editors for code and data. *Proceedings of the 18th annual ACM symposium on User interface software and technology* (2005), 3–12.

[24] Kölling, M., Brown, N.C.C. and Altadmri, A. 2015. Frame-Based Editing: Easing the Transition from Blocks to Text-Based Programming. *Proceedings of the Workshop in Primary and Secondary Computing Education* (New York, NY, USA, 2015), 29–38.

[25] Lewis, C.M. 2010. How programming environment shapes perception, learning and goals: Logo vs. Scratch. *Proceedings of the 41st ACM Technical Symposium on Computer Science Education* (New York, NY, 2010), 346–350.

[26] Maloney, J.H., Peppler, K., Kafai, Y., Resnick, M. and Rusk, N. 2008. Programming by choice: Urban youth learning programming with Scratch. *ACM SIGCSE Bulletin*. 40, 1 (2008), 367–371.

[27] Maloney, J.H., Resnick, M., Rusk, N., Silverman, B. and Eastmond, E. 2010. The Scratch programming language and environment. *ACM Transactions on Computing Education (TOCE)*. 10, 4 (2010), 16.

[28] Matsuzawa, Y., Ohata, T., Sugiura, M. and Sakai, S. 2015. Language Migration in non-CS Introductory Programming through Mutual Language Translation Environment. *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (2015), 185–190.

[29] Price, T.W. and Barnes, T. 2015. Comparing Textual and Block Interfaces in a Novice Programming Environment. (2015), 91–99.

[30] Resnick, M., Silverman, B., Kafai, Y., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E. and Silver, J. 2009. Scratch: Programming for all. *Communications of the ACM*. 52, 11 (Nov. 2009), 60.

[31] Robins, A., Rountree, J. and Rountree, N. 2003. Learning and teaching programming: A review and discussion. *Computer Science Education*. 13, 2 (2003), 137–172.

[32] Roque, R., Kafai, Y. and Fields, D. 2012. From tools to communities: Designs to support online creative collaboration in Scratch. *Proceedings of the 11th International Conference on Interaction Design and Children* (2012), 220–223.

[33] Sherin, B.L. 2001. A comparison of programming languages and algebraic notation as expressive languages for physics. *International Journal of Computers for Mathematical Learning*. 6, 1 (2001), 1–61.

[34] Shneiderman, B. 1983. Direct manipulation: a step beyond programming languages. *Computer*. 16, 8 (1983), 57–69.

[35] Steele, C.M. and Aronson, J. 1995. Stereotype threat and the intellectual test performance of African Americans. *Journal of personality and social psychology*. 69, 5 (1995), 797.

[36] Stefik, A. and Siebert, S. 2013. An Empirical Investigation into Programming Language Syntax. *ACM Transactions on Computing Education*. 13, 4 (Nov. 2013), 1–40. DOI:https://doi.org/10.1145/2534973.

[37] Tempel, M. 2013. Blocks Programming. *CSTA Voice*. 9, 1 (2013).

[38] Tew, A.E. and Guzdial, M. 2011. The FCS1: a language independent assessment of CS1 knowledge. *Proceedings of the 42nd ACM technical symposium on Computer science education* (2011), 111–116.

[39] Weintrop, D., Afzal, A., Salac, J., Francis, P., Li, B., Shepherd, D.C. and Franklin, D. 2018. Evaluating CoBlox: A Comparative Study of Robotics Programming Environments for Adult Novices. *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems* (Montreal QC, Canada, 2018), 366:1–12.

[40] Weintrop, D. and Holbert, N. 2017. From Blocks to Text and Back: Programming Patterns in a Dual-Modality Environment. *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education* (New York, NY, USA, 2017), 633–638.

[41] Weintrop, D., Killen, H. and Franke, B. 2018. Blocks or Text? How programming language modality makes a difference in assessing underrepresented populations. *Proceedings of the International Conference on the Learning Sciences 2018* (London, UK, 2018), 328–335.

[42] Weintrop, D. and Wilensky, U. 2017. Comparing Block-Based and Text-Based Programming in High School Computer Science Classrooms. *ACM Transactions on Computing Education (TOCE)*. 18, 1 (Oct. 2017), 3. DOI:https://doi.org/10.1145/3089799.

[43] Weintrop, D. and Wilensky, U. 2015. To Block or Not to Block, That is the Question: Students' Perceptions of Blocks-based Programming. *Proceedings of the 14th International Conference on Interaction Design and Children* (New York, NY, USA, 2015), 199–208.

[44] Weintrop, D. and Wilensky, U. 2015. Using Commutative Assessments to Compare Conceptual Understanding in Blocks-based and Text-based Programs. *Proceedings of the Eleventh Annual International Conference on International Computing Education Research* (New York, NY, USA, 2015), 101–110.

[45] Werner, L., Denner, J., Campe, S. and Kawamoto, D.C. 2012. The fairy performance assessment: measuring computational thinking in middle school. *Proceedings of the 43rd ACM technical symposium on Computer Science Education* (2012), 215–220.

[46] Wilensky, U. and Papert, S. 2010. Restructurations: Reformulating knowledge disciplines through new representational forms. *Proceedings of the Constructionism 2010 conference* (Paris, France, 2010).