Computers & Education 142 (2019) 103646

Contents lists available at ScienceDirect

Computers & Education

journal homepage: www.elsevier.com/locate/compedu

# Transitioning from introductory block-based and text-based environments to professional programming languages in high school computer science classrooms

David Weintrop[a,*], Uri Wilensky[b]

[a] College of Education, College of Information Studies, University of Maryland, College Park, USA
[b] Center for Connected Learning and Computer-based Modeling, Northwestern University, USA

ARTICLE INFO

*Keywords:*
Evaluation of CAL systems
Interactive learning environments
Programming and programming languages
Secondary education
Teaching/learning strategies

ABSTRACT

Block-based programming languages are becoming increasingly common in introductory computer science classrooms across the K-12 spectrum. One justification for the use of block-based environments in formal educational settings is the idea that the concepts and practices developed using these introductory tools will prepare learners for future computer science learning opportunities. This view is built on the assumption that the attitudinal and conceptual learning gains made while working in the introductory block-based environments will transfer to conventional text-based programming languages. To test this hypothesis, this paper presents the results of a quasi-experimental classroom study in which programming novices spent five-week using either a block-based or text-based programming environment. After five weeks in the introductory tool, students transitioned to Java, a conventional text-based programming language. The study followed students for 10 weeks after the transition. Over the course of the 15-week study, attitudinal and conceptual assessments were administered and student-authored programs were collected. Conceptual learning, attitudinal shifts, and changes in programming practices were analyzed to evaluate how introductory modality impacted learners as they transitioned to a professional, text-based programming language. The findings from this study build on earlier work that found a difference in performance on content assessments after the introductory portion of the study (Weintrop & Wilensky, 2017a). This paper shows the difference in conceptual learning that emerged after five weeks between the block-based and text-based conditions fades after 10 weeks in Java. No differences in programming practices were found between the two conditions while working in Java. Likewise, differences in attitudinal measures that emerged after working in the introductory environments also faded after 10 weeks in Java, resulting in no difference between the conditions after 15 weeks. The contribution of this work is to advance our understanding of the benefits and limits of block-based programming tools in preparing students for future computer science learning. This paper presents the first quasi-experimental study of the transfer of knowledge between block-based and text-based environments in a high school setting. The lack of significant differences between the two introductory programming modalities after learners transition to professional programming languages is discussed along with the implications of these findings for computer science education researchers and educators, as well as for the broader community of researchers studying the role of technology in education.

* Corresponding author.
*E-mail address:* weintrop@umd.edu (D. Weintrop).

https://doi.org/10.1016/j.compedu.2019.103646
Received 1 August 2018; Received in revised form 30 July 2019; Accepted 4 August 2019
Available online 07 August 2019
0360-1315/ © 2019 Elsevier Ltd. All rights reserved.

## 1. Introduction

Block-based programming is quickly becoming the way that younger learners are being introduced to the field of computer science (Bau, Gray, Kelleher, Sheldon, & Turbak, 2017). Led by the popularity of tools such as Scratch (Resnick et al., 2009), Blockly (Fraser, 2015), and Alice (Cooper, Dann, & Pausch, 2000), millions of kids are engaging with programming through drag-and-drop graphical tools. For example, Code.org's Hour of Code initiative, which includes dozens of activities that incorporate block-based programming, has recently surpassed 500 million "hours served" and has reached learners in every country on the planet (Code.org, 2017). This highlights how the excitement around computer science is a global phenomenon. Due in part to the success of such tools and initiatives at engaging novices in programming, block-based programming environments are increasingly being incorporated into curricula designed for high school computer science classrooms. Examples of such curricula include Exploring Computer Science (Goode, Chapman, & Margolis, 2012), the Computer Science Principles project (Cuny, 2015), and Code.org's curricular offerings (Code.org Curricula, 2019).

Many uses of block-based tools in formal educational contexts presuppose that such tools will help prepare students for later instruction in text-based languages (Armoni, Meerbaum-Salant, & Ben-Ari, 2015; Brown et al., 2016; Dann, Cosgrove, Slater, Culyba, & Cooper, 2012). This transition is often a part of the larger computer science trajectory where block-based introductory courses are intended to prepare students for the transition to professional, text-based languages. This can be seen in how the transition has been studied to date (e.g. Armoni et al., 2015; Dann et al., 2012; Powers, Ecott, & Hirshfield, 2007). This assumption was also echoed by the high school students who participated in the study presented in this work, who made statements such as "*[block-based programming] is a good start, then once we know the commands and everything, we can move on to Java*" and "*[block-based programming] is getting us ready for what we're going to be doing*". While work has been done focusing on learning that happens while using block-based tools (e.g. Franklin et al., 2017; Grover & Basu, 2017; Weintrop & Wilensky, 2015a), less work has rigorously tested the transition from block-based introductory tools to text-based languages in formal settings (Blikstein, 2018; Shapiro & Ahrens, 2016). A systematic review of the literature on the role of visual programming concluded that there is uncertainty concerning the effectiveness of block-based languages when looking beyond introductory contexts (Noone & Mooney, 2018). This question is of great importance given the growing role of block-based tools in K-12 education around the world and their impact on the teaching and learning of computer science (Blikstein, 2018; Caspersen, 2018).

This paper seeks to understand if and how the modality used (block-based versus text-based) prepares learners for conventional text-based languages. This line of inquiry is consequential for both the research community as it is an open question that can inform future research on design and learning, as well as practitioners who are tasked with making decisions around learning environments and pedagogy in their classrooms every day. More specifically, this paper answers the following research question:

In high school introductory computer science classes, how does the modality used for introductory programming instruction (block-based versus text-based) impact learners when they transition to a professional text-based programming language?

This paper presents the results of a quasi-experimental study designed to answer this question. The study took place in two high school computer science classrooms and compares isomorphic block-based and text-based programming environments. Students spent five weeks working in either a block-based or text-based version of the same introductory programming environment before transitioning to Java. The same teacher taught both classes and students in each condition used the same curriculum and had the same time-on-task. The findings from the first five weeks of the study are reported in (Weintrop & Wilensky, 2017a). This work is a continuation of that paper, specifically focusing on what happened after leaving the introductory environments and moving to Java. To understand how the design of introductory tools prepare learners for programming in professional programming languages, we present comparative outcomes of content assessments, attitudinal surveys, and investigate programming practices that emerged after the transition to Java.

## 2. Prior work

### 2.1. Block-based programming

Block-based programming is a visual programming paradigm that utilizes a programming-primitive-as-puzzle-piece metaphor to make the act of programming more accessible and intuitive for novices (Bau et al., 2017; Good, 2018). The block-based programming approach is becoming increasingly widespread. Duncan, Bell, and Tanimoto (2014) reviewed 47 introductory programming environments and found 28 of the environments used the block-based approach to programming, including 19 of the 24 environments designed for learners under the age of 8. Writing a program in a block-based programming environment takes the form of snapping together commands by dragging-and-dropping them next to each other. Block-based programming environments include several features designed to facilitate the act of programming (Maloney, Resnick, Rusk, Silverman, & Eastmond, 2010; Tempel, 2013). For example, the visual depiction of a block provides cues denoting how and where a given command can be used. Those visual cues are also used as a means of enforcing syntactic rules, preventing incompatible or incorrect statements from being combined to create invalid statements. In this way, the block-based modality prevents syntax errors during program construction but retains the practice of authoring programs instruction-by-instruction. Block-based programming environments also support the programmer by presenting the available commands in easily-browsed and logically organized drawers, a feature identified by learners as easing the barrier to programming (Weintrop & Wilensky, 2015b). Collectively, these features provide a rich web of supports for novices to draw on making them a compelling way to introduce novices to programming (Kölling & McKay, 2016; Weintrop & Wilensky, 2017b).

Research looking at the use of block-based programming environments in introductory K-12 educational contexts is showing it to

be an effective way to introduce novices to foundational programming concepts. At the upper elementary level (ages 10–14), block-based tools are an effective way to make programming concepts accessible to younger learners when provided a developmentally appropriate curriculum (Franklin et al., 2017; Hill, Dwyer, Martinez, Harlow, & Franklin, 2015; Howland & Good, 2014; Meerbaum-Salant, Armoni, & Ben-Ari, 2010). After teaching a computer science curriculum to upper elementary school learners, Grover at al. (2015) found block-based programming to be a productive way to introduce learners to foundational computing concepts and develop important computational thinking skills such as algorithmic thinking. Comparative studies between block-based and text-based environments with students in this age group found that learners perform comparably or better in the block-based condition and complete assignments more quickly (Lewis, 2010; Matsuzawa et al., 2015; Price & Barnes, 2015, pp. 91–99). At the high school level, a comparative study found that students perform better on concept assessments after working in block-based tools compared to isomorphic text-based alternatives after five weeks of instruction (Weintrop & Wilensky, 2017a). Collectively, this work supports the decision of using block-based programming modality in K-12 classrooms.

Along with conceptual gains, block-based programming environments have been shown to have positive impacts with respect to motivation, attitudes, and engagement for K-12 learners. This has been found across a number of programming environments including Scratch (Malan & Leitner, 2007, pp. 223–227; Resnick et al., 2009; Ruf, Mühling, & Hubwieser, 2014, pp. 50–59), Snap! (Garcia, Harvey, & Barnes, 2015), and Alice (Kelleher, Pausch, & Kiesler, 2007; Kelleher & Pausch, 2007). It is also important to note that the block-based approach to programming has been successful at engaging novice programmers from historically under-represented populations (Kelleher et al., 2007; Maloney, Peppler, Kafai, Resnick, & Rusk, 2008; Tangney, Oldham, Conneely, Barrett, & Lawlor, 2010; Wilson & Moffat, 2010).

### 2.2. Dual-modality and Bi-directional programming environments

One active area of research relevant to the current study is design work exploring ways to combine block-based programming with features of conventional text-based representations of computational ideas. The idea with this approach is to draw on multiple modalities to support learners making meaning of programming concepts and authoring successful programs (Good & Howland, 2017; Weintrop & Holbert, 2017). This is part of the larger conversation around the long-term role of block-based programming in education and beyond (Shapiro & Ahrens, 2016; Weintrop, 2019). We divide this body of research into two categories: dual-modality environments, which combine features of block-based and text-based tools in a single interface, and bi-directional environments that allow learners to move back-and-forth between block-based and text-based presentations of the program. These environments are pertinent to this study as they represent alternatives to the notion of block-based programming serving as a predecessor to text-based programming.

There are a growing number of bi-directional programming environments that are gaining popularity in computer science education circles. The Droplet editor (Bau, 2015), which is used in Code.org's AppLap and the Pencil Code environment, provides a button allowing the learner to choose whether they want to work in a block-based interface or text-based interface. Clicking the transition button begins an animation where the learner can watch their code morph from one modality to the other, reinforcing the equality of the two presentations of code. The Droplet editor currently supports JavaScript, HTML, and CoffeeScript, with a Python implementation currently under development (Blanchard, 2017). Further implementations of this approach include BlockPy, a Python-based tool focusing on data science contexts (Bart, Tibau, Kafura, Shaffer, & Tilevich, 2017), Tiled Grace, an overlay on top of the Grace programming language supporting block-based interactions (Homer & Noble, 2017), GP, a reimagining of the Scratch environment designed to be more general purpose (Mönig, Ohshima, & Maloney, 2015), and the work of Matsuzawa, Ohata, Sugiura, and Sakai (2015) who created a bi-directional Java environment that was one of the first to empirically study this approach and show its promise. Research is revealing some of the ways that students take advantage of the bi-directionally of these environments, such as shifting from text to blocks in order to introduce new commands to a program (Weintrop & Holbert, 2017).

Unlike bi-directional environments, the dual-modality approach blends features of block-based and text-based programming into a single editor. The idea with this strategy is to try and leverage the strengths of both modalities at the same time. This can be seen in the Frame-based editor build for Greenfoot's Stride language, which is designed to be a keyboard-driven approach to block-based programming (Kölling, Brown, & Altadmri, 2017). Research on this approach has shown students have similar learning gains as those in traditional environments, but progress more quickly and have fewer issues related to language syntax (Price, Brown, Lipovac, Barnes, & Kölling, 2016). Further research looking at novices working in dual-modality environments has revealed that learners take advantage of aspects of both modalities as they work, making it distinct from working in either a fully blocks-based or fully text-based environment (Weintrop & Wilensky, 2018). The active development of such environments speaks to the current relevance of the proposed study, which seeks to understand how block-based environments do and do not support learners in their transition to text-based languages.

### 2.3. From block-based to text-based programming

As block-based programming environments have grown in popularity in formal educational settings, a question of increasing consequence is whether or not the approach is effective at preparing learners for future, text-based programming languages. Early work on graphical (but not block-based) programming found little evidence of successful transfer in novices when moving to text-based programming languages meaning novices still struggled with basic programming tasks in the new language despite prior programming success with graphical tools (Scholtz & Wiedenbeck, 1990; Wiedenbeck, 1993). Similar outcomes have been documented using contemporary block-based environments, with numerous case studies documenting challenges associated with the

transition (e.g. Cliburn, 2008; Garlick & Cankaya, 2010; Powers et al., 2007). Kölling, Brown, and Altadmri (2015) document 13 distinct issues related to this transition, ranging from the need to memorize syntax to the shift to typing in commands character-by-character in place of blocks that treat commands as atomic objects.

At the same time, there are studies showing that gains made in block-based tools do prepare learners for later text-based programming. Armoni et al. (2015) conducted a longitudinal study to investigate this topic. Their approach was to look at the performance of students in a high school text-based programming class that was comprised of some students who had taken a Scratch programming class in middle school and other students with no prior programming experience. The researchers then used the students' performance in the high school programming course to assess whether or not the Scratch middle school experience helped students in this later course. The resulting analysis reported little quantitative difference in their performances on assessments but did identify some specific content areas where the students with earlier Scratch experience out-performed their peers (e.g. iterating programming constructs). The authors also found that students with prior Scratch experience reported higher levels of self-efficacy and motivation to learn to program. Grover, Pea, and Cooper (2015) used a preparation for future learning lens to show that a block-based curriculum can prepare learners for text-based programming, including results showing learners performing well on conceptual multiple-choice questions posed in text-based languages after working through a Scratch-based curriculum. Using a version of the Alice designed to support the transition and using a pedagogical strategy that emphasized this transition, Dann et al. (2012) found students performed better in future Java courses after completing the transfer-focused curriculum. Continuing scholarship on the transition is revealing promise in this approach to helping learners make the transition (e.g. Saito, Washizaki, & Fukazawa, 2016; Tabet, Gedawy, Alshikhabobakr, & Razak, 2016). These studies suggest that block-based tools can help learners in future text-based contexts, especially as it relates to conceptual knowledge and the ability to author successful programs. However, the lack of a controlled comparison or quasi-experimental design in this prior work prevents these authors from making strong claims on the role of modality in supporting this transition independent of curriculum or pedagogy. It is this question, and the associated methodological gaps, that the present work seeks to address.

## 3. Materials and methods

### 3.1. Study design and data collection strategy

This paper seeks to answer the following research question: In high school introductory computer science classes, how does the modality used for introductory programming instruction (block-based versus text-based) impact learners when they transition to a professional text-based programming language? To answer it, we conducted a quasi-experimental study during the first 15 weeks of the school year in two introductory programming classes. The two classes followed the same curriculum but used different versions of the same programming environments during the first portion of the course. For the first five weeks of the school year, one class used a block-based version of the environment and the other class used a text-based version of the same environment. Further details about the environment are presented in the next section. After the five-week introduction, both classes transitioned to the Java programming environment. Once in Java, both classes used the same programming environment and again followed the same curriculum. The study design had students take a content assessment and attitudinal survey at the outset, after five weeks when the transition to Java took place and then again at the conclusion of the 15 weeks. The content assessment and attitudinal assessments were administered on consecutive days during class time and took roughly 20 min to complete each. A detailed analysis of comparative findings from the first five weeks of the study can be found in (Weintrop & Wilensky, 2017a) and full versions of both instruments used in this study can be found in the appendices of (Weintrop, 2016).

For the content assessment, students were asked to complete a customized version of the Commutative Assessment designed specifically for the study (Weintrop & Wilensky, 2015a). The assessment is comprised of 30 multiple-choice questions on the programming concepts covered during the introductory portion of the course (including conditional and iterative logic, variables, and functions). The assessment also included questions related to overall program comprehension and non-programming questions on algorithms. The defining feature of the Commutative Assessment is that each of the short programs can be presented in either Snap! Blocks, Pencil Code blocks, or Pencil Code text (Fig. 1a, b, and 1c respectively). The comprehension questions ask students to select the correct description of the presented program, so rather than figuring out just the output of the program, students must derive the purpose of the set of instructions. Note, we acknowledge the shortcomings of multiple-choice questions for program comprehension questions (Simon and Snowdon, 2014) but include them in the analysis as there were a small portion of the overall assessment and were held constant between groups making them still useful for our comparative questions. Each version of the assessment asked


**Fig. 1.** The three forms programs may take in the Commutative Assessment., (a) Snap!, (b) Pencil Code Blocks, (c) Pencil Code Text.

students to answer questions in all three of the programming modalities. The assessment was designed such that during each administration of the assessment students answered questions for every topic in all three modalities. Further, three versions of assessment were constructed with different patterns of question-modality pairs and were given at each of the three time intervals. This resulted in a counterbalanced design where every student answered every question in each of the three modalities over the course of the 15 weeks, ensuring an even distribution of concept, condition, and modality.

Along with the content assessment, students completed an attitudinal survey at the beginning, middle, and conclusion of the study. The three versions of the survey administered during the study were largely the same except for changes in tense (past/future) and additional questions added to the Mid and Post surveys asking students to reflect on their experiences in the class. The survey was based on items from the Georgia Computes project (Bruckman et al., 2009, pp. 86–90) and the Computing Attitudes Survey which has been validated and is widely used in computing education research studies (Dorn & Elliott Tew, 2015; Tew, Dorn, & Schneider, 2012). The survey was comprised of 10-point Likert scale questions as well as short response questions. In the analysis presented in this paper, we focus on questions associated with confidence and enjoyment and a standalone question related to interested in future computer science courses.

Finally, as part of this study, we recorded the programs students wrote and the error message they received while compiling their programs. When programming in Java, before a program can be run it must be compiled using the `javac` command. If there is an error in the program, a call to `javac` will respond with an error, preventing the program from being run. For this study, we recorded every call to `javac` made by the participants, capturing the contents of the program and the output of the `javac` call (including any errors if present). To accomplish this, we added logic to the console students used to run their programs. To record these events, we developed a tool that wrapped the students' compilation command with a script that executes the compilation while also recording the contents of the student's program and the resulting compiler output and sends both to a remote server. This whole process is invisible to the student. This approach to logging student programs builds on earlier work looking at compilation events as a means to gain insight into learners' emerging programming practices (Jadud & Henriksen, 2009). The specific implementation used in this work was informed by the Git Data Collection project (Danielak, 2014).

### 3.2. Pencil.cc and the introductory curriculum

This research study is built around students learning to program in the same programming environment but using different modalities – either block-based or text-based. To carry out this study, a modified version of Pencil Code was created called Pencil.cc. Pencil Code is an online programming environment that allows users to freely move back-and-forth between text-based (Fig. 2a) and block-based (Fig. 2b) versions of their programs (Bau, Bau, Dawson, & Pickens, 2015). In supporting this bi-directionality, the two programming modalities are isomorphic, meaning that anything that can be done in one interface can also be done in the other. Unlike Pencil Code, Pencil.cc prevents learners from moving between the two modalities, instead, learners either use only the block-based interface or only the text-based interface. Thus, for the duration of the five-week study, students were introduced to programming using either a block-based version of Pencil.cc or a text-based version of Pencil.cc. This means students in one class programmed via the drag-and-drop mechanism supported by the block-based interface while the other class authored programs by typing in commands character-by-character. Aside from the programming modality, everything else about the two versions of the programming environment is identical, including the programming language (including keywords and syntax), the visual execution environment, and the programming capabilities and other environmental scaffolds. For both versions of Pencil.cc, the underlying programming language used was CoffeeScript. CoffeeScript was chosen as it is syntactically light, has an active professional user base, and it is sufficiently different from Java so as to keep the transition between environments significant.

The block-based interface of Pencil.cc includes many of the features of block-based programming that research has identified that learners perceive as being productive for programming (Weintrop & Wilensky, 2015b). This includes a conceptually organized and browsable palette of blocks (the left-most portion of Fig. 2b), visual cues as to how and where blocks can be used, and the drag-and-drop compositional mechanism. Students in the text-based condition of the study used a text-editor that includes numerous scaffolds

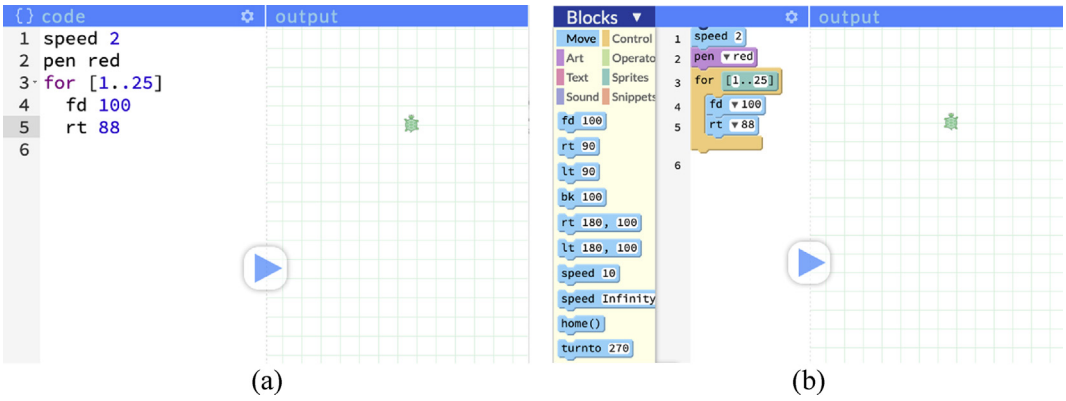


**Fig. 2.** The text-based (a) and block-based (b) versions of Pencil.cc used for this study.

standard to programming authoring tools including syntax highlighting and basic compile-time error checking. The goal in creating Pencil.cc was to isolate the programming modality to understand if and how the way students are initially introduced to programming (i.e. block-based vs. text-based) affects their ability to transition to conventional text-based languages.

The introductory curriculum students followed for the first five weeks was a modified version of the Beauty and Joy of Computing (Garcia et al., 2015). Additional activities were created to take advantage of features of Pencil.cc that were grounded in the Constructionist programming tradition (Harvey, 1997; Papert, 1980). In creating the curriculum, an effort was made to ensure student were given creative freedom within each assignment. As such, student solutions varied widely within the assignments. The five-week curriculum covered the following topics: variables, conditional logic, iterative logic, and functions. Each content area included visual assignments that asked students to control an on-screen turtle (in the spirit of Logo and Scratch activities) as well as text processing activities with no graphical component. The types of assignments in the curriculum were balanced to not privilege one condition over the other in the case that one modality was more conducive to a particular type of activity. The curriculum used for the study is available in Appendix A of (Weintrop, 2016).

#### *3.2.1. Introductory Java materials*

After the five-week introduction, students in both conditions transition to Java. All students used the same basic text editor for their Java programming assignments, which did not include common programming editor features like syntax highlighting or autocompletion. The teacher felt it was pedagogically valuable to have students work in a basic text editor, which she had successfully used in this course in prior years. The course followed the Java Concepts: Early Objects textbook (Horstmann, 2012). During the ten weeks of the Java portion of the study, students encountered basic input/output, variables, data types, objects, and functions. While there is not complete content overlap between the introductory curriculum and the first ten weeks in Java, there are concepts that were encountered in both, notably variables and functions.

#### *3.2.2. Setting and participants*

This study was carried out in an urban, public school in the American Midwest that serves almost 4,000 students. The school is racially and socio-economically diverse (44% Hispanic, 33% White, 10% Asian, 9% Black, and 4% multiracial/other; 58.6% of students are from economically disadvantaged households). The school is a selective enrollment institution, meaning it is designed for academically well-performing students who must do well on a written test to be admitted. The school district has put measures in place to ensure the makeup of selective enrollment schools are representative of the urban population from which they draw in that students are admitted based on their test performance relative to other students at their current school rather than to other students across the city. The result of this decision is a more diverse (both socioeconomically and geographically) student body than alternative approaches for deciding admissions. Working in a selective enrollment school was not ideal for this work and introduces some limitations but was necessary as few public schools have multiple sections of the same computer science course or a teacher experienced enough to be willing to teach different classes using variations of the same programming tool. The student population that participants were drawn from captures a representative set of high-achieving learners.

The quasi-experimental design was carried out in two sections of an existing Introduction to Programming course. In prior years, the course began teaching Java on day one. For this study, the introduction to Java was delayed until the sixth week of class, with the first five weeks being spent using Pencil.cc and the custom-designed curriculum discussed above. Each of the two classes had 30 students, each of whom were assigned a laptop to use for the course. The two classes were taught by the same teacher in the same classroom in back-to-back periods allowing us to control for teacher and environmental effect. The teacher received her undergraduate degree in technical education and corporate training and was in her eighth year of teaching (third at that school).

The Introduction to Programming class is offered as an elective meaning students choose to enroll in the class. Historically the class attracts students from a variety of racial backgrounds and usually has more male students enroll than female students. This study included a total of 60 students with all students in the two classes included in the set of potential participants. The choice of two classrooms of 30 each was chosen as 30 is often held as the minimum number of participants for statistical analysis which gives us the ability to analyze conditions independently (Cohen, Manion, & Morrison, 2007). Further, the school only offered 3 sections of the course, all of which were involved in research studies and including a second school would have introduced new confounds to the study design and analysis. The self-reported racial breakdown of students in the class was as follows: 41% White, 27% Hispanic, 11% Asian, 11% Multiracial, and 10% Black. Among participants, a language other than English is spoken in 47% of homes. The classes were made up of 49 male students (25 in Blocks, 24 in Text) and 11 female students (5 in Blocks, 6 in Text). The two classes included students from all four high school grades (9 freshmen, 9 sophomores, 16 juniors, and 26 seniors). The gender disparity in these classrooms is problematic but recruiting students to enroll in the classes used in the study was beyond the control of the researchers.

## 4. Results

This paper seeks to understand how introductory programming modality (block-based versus text-based) impacts learners when they move on to learning a professional text-based programming language. For this work, we operationalize “impact” by looking at three interrelated dimensions of learning to program: conceptual understanding, attitudes towards programming, and programming practices. In focusing on these three aspects, we seek to understand how the knowledge, dispositions, and practices that developed using introductory tools carry over to learning to program with a professional text-based language. We use these three aspects as a means to understand different dimensions of the complex act of learning to program. As such, this section presents three different ways of understanding how introductory modality impacts the novice learning experience as they move from introductory to

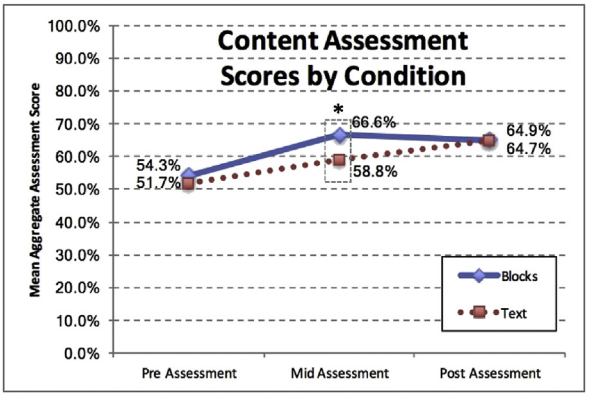


**Fig. 3.** The mean scores for students in the two conditions on the three administrations (Pre, Mid, and Post) of the Commutative Assessment.

professional tools.

This section begins with a presentation of the findings of the learners conceptual understanding of programming content through the analysis of student performance on the Commutative Assessment. Next, we present a systematic analysis of responses to the attitudinal surveys administered. Finally, we present an analysis of student programming practices based on an analysis of learner's calls to `javac`. Throughout this section, we briefly report findings from the previously published first five weeks of the study to serve as context but focus specifically on results from week 5 onwards. Additionally, given the multidimensional nature of the analysis, we provide some short discussions of results specific to each analysis here before providing an overarching analysis across the different data sources in the discussion section that follows.

### *4.1. Learning outcomes*

This section presents an analysis of the Pre, Mid, and Post content assessments scores by condition. Fig. 3 shows student scores at each of the three administrations.

We begin by presenting previously published results looking at scores on the content assessment administered at the outset of the school year and the midpoint of the study (Weintrop & Wilensky, 2017a). By condition, the mean scores were 51.7% (*SD* 14.5%) for Text and 54.3% (*SD* 12.2%) for the Blocks condition, a difference not found to be statistically significant: $t(57) = 0.78$, $p = 0.44$, $d = 0.20$ (note: calculations showing participant numbers less than 60 is a result of student absences where no make-up administration was possible). This analysis shows there to be no difference between the two classes with respect to prior programming knowledge at the outset of the study. After working in the introductory environment for 5 weeks, a difference between the two conditions emerged. Looking at the Mid assessment, students in the Text condition had a mean score of 58.8% (*SD* 14.6%) and students in the Blocks condition scored an average of 66.6% (*SD* 13.4%). A *t*-test shows this difference to be statistically significant ($t(53) = 2.03$, $p = 0.04$, $d = 0.58$). This means that after 5 weeks, students learning to program in a block-based environment performed significantly better on a content assessment than peers using an isomorphic text-based environment.

After the Mid assessment, students immediately transitioned to Java and spent the next 10 weeks programming in Java before taking the Commutative Assessment for a third and final time. On the Post content assessment, there is no statistical difference found between the two conditions ($t(57) = 0.014$, $p = .99$), with the Blocks students having an average of 64.9% (*SD* 13.5%) and students coming from the Text condition scoring 64.7% (*SD 14.0%)*. If we look at how student performance changed between the Mid and Post administrations and compare it between the two conditions, we find the difference in gains made by students between the two conditions to be statistically significant ($t(52) = 2.58$, $p = .01$, $d = 0.70$).

This analysis reveals that students in the block-based programming condition achieved greater gains during the five-week introductory period with respect to performance on the Commutative Assessment. After the transition to Java, students in the Blocks conditions did not improve, while the Text condition saw another incremental improvement. This means that over the course of the 15-week study, all of the students started and ended in the same place on the content assessment, however, the two conditions took different paths along the way.

One possible explanation is that there was little transfer between environments for students coming from the block-based condition, so their performance plateaued, while students who had spent the previous five weeks working in the text-based introductory environment experienced some transfer that helped them continue developing their conceptual understanding of programming concepts as they moved into Java. A second possible explanation of this finding is that there is a ceiling effect for learners and that the Blocks condition reached that ceiling faster than the Text condition. In other words, learners in the environment that enabled drag-and-drop composition were able to more quickly understand the concepts at hand, while the Text condition took longer to make sense of the activity of programming before reaching the ceiling associated with the curriculum students worked through. The finding that block-based learning environments allow students to learn more quickly has been shown in some small studies in informal environments (Price & Barnes, 2015, pp. 91–99), so this suggests this may be a larger, more robust phenomenon.

### 4.2. Attitudinal findings

Part of understanding and evaluating the impact of working in different modalities during the introductory portion of the course is investigating how the attitudes and perceptions of programming that formed during their use persisted or changed as students moved on to Java. As previously mentioned, in this section, we present the results of our analysis of two composite attitudinal measures (confidence and enjoyment) and a standalone question related to interested in future computer science courses. The decision to focus on these specific dimensions of attitude is due to their association with a learner's likelihood of pursuing future computer science learning opportunities, especially as it relates to learners from historically underrepresented population in computing (American Association of University Women, 1994; Margolis & Fisher, 2003; Tavani & Losh, 2003). Additionally, confidence and enjoyment are often used in studies investigating the design of programming environments and learning (e.g. Bishop-Clark, Courte, & Howard, 2006; McDowell, Werner, Bullock, & Fernald, 2006; Rodriguez Corral et al., 2019). When analyzing changes in scores over time within the same set of students (e.g. between the Mid and Post surveys for the Blocks condition), a Wilcoxon Signed Rank test (reported as Z statistic) is performed. The ordinal nature of the Likert responses and non-parametric nature of the test make this an appropriate test to use for these paired sample calculations. In cases where the analysis is comparing the two conditions to each other, a Wilcoxon Rank Sum test (reported as a U statistic) is performed. This test is used because the two samples are independent and the underlying data is ordinal and non-parametric (Fay & Proschan, 2010). Note, the figures in this section are all presented on the same y-axis scale but do not all cover the same range, meaning they can be compared relative to each other but the meanings of positions on the charts differ. It is also important to note that the y-axes of figures presented in this section do not start at zero, this was done to make quantitative differences more legible.

#### 4.2.1. Confidence in programming ability

The first attitudinal dimension discussed is students' perceived confidence in their programming ability. The composite confidence score is the combination of the two Likert statements: I will be good at programming and I will do well in this course (note: the tense of the statements changed between administrations). These questions have Cronbach's α scores of 0.82 on the Pre survey, 0.80 on the Mid survey and 0.88 on the Post survey, which meet the 0.8 threshold often cited as the minimum level of acceptability for research purposes (Streiner, 2003). Fig. 4a shows the aggregated confidence measure at the Pre, Mid and Post points in time.

Looking at the three distinct points at which the survey was administered, we find no significant difference in confidence between the conditions (Pre: $U = 353.5$, $p = .30$; Mid: $U = 395$, $p = .78$; Post: $U = 307.5$, $p = .08$). Running Wilcoxon signed ranked tests for changes within groups between time points, we only find a significant difference in the Blocks condition between the Mid and Post survey, which shows learners who spent 5 weeks working in the block-based condition to be significantly less confident after spending 10 weeks programming in Java ($Z = 46$, $p = .05$). Note that despite there being a numerical difference between the conditions at the outset of the study, the difference was not statistically significant, nor was the change in confidence between the Pre and Mid administrations of the survey. Taken together, the data show that, comparatively, the modality did not have a significant impact on students' confidence. Linking these results with the findings from the previous section on content performance, we see students in the Text condition showing a slight (though not significant) growth in confidence at the same time their test scores are improving, while the Blocks students see a decrease in confidence alongside a decrease in scores on the content assessment. The conclusion to be drawn from this analysis is that modality alone does not seem to affect high school learners' confidence in their programming ability.

Looking just at the trajectory for the Blocks conditions, the increase in confidence for students could explain other findings showing an increased retention for students using these types of graphical tools in their first computer science course (Cliburn, 2008; Johnsgard & McDonald, 2008), but does potentially call into question the effectiveness of such an approach for preparing students for future learning of computer science as the gains with respect to confidence do not persist.

#### 4.2.2. Enjoyment of programming

The second attitudinal dimension analyzed from the survey seeks to understand if students enjoyed programming and if so, how it

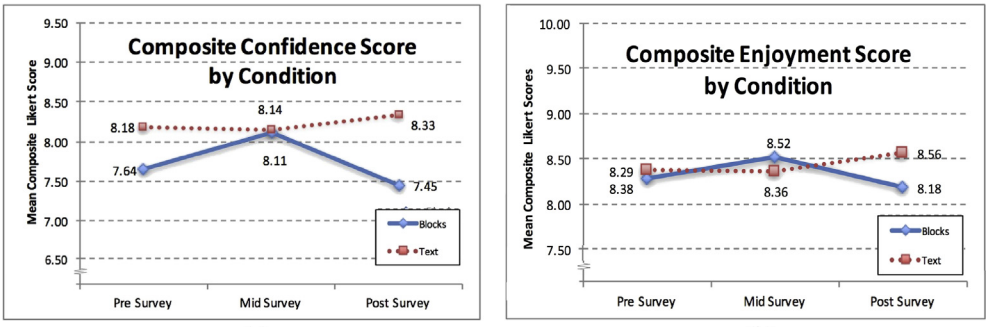


**Fig. 4.** Composite scores of students' confidence (a) and enjoyment (b) in programming at three points in the study.

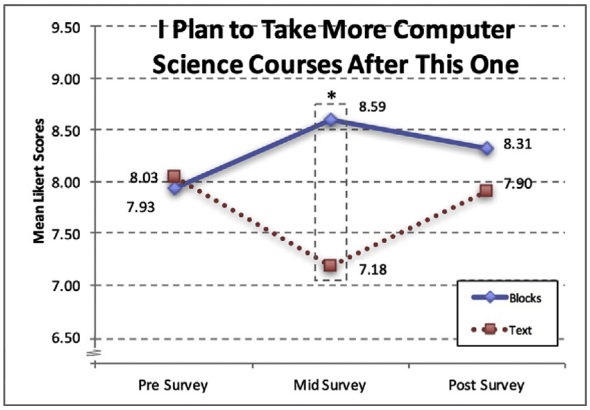


**Fig. 5.** The mean responses scores, grouped by condition, for the statement: I plan to take more computer science courses after this one.

differed by condition both during their time using the introductory tools and their time in Java. The aggregate enjoyment score is a composite of responses to the following three questions: Programming is Fun, I like programming, and I am excited about this course. A Cronbach's Alpha test was run on these three questions and found a sufficient level of correlation (Pre: $\alpha = 0.79$, Mid: $\alpha = 0.84$, Post: $\alpha = 0.89$).

Looking at the changes from Pre to Mid and Mid to Post, shown in Fig. 4b, there is very little quantitative change with no statistically significant differences emerging between time periods within the groups or at the same time period across the groups. This lack of difference leads to the conclusion that modality does not affect perceived enjoyment in high school learners. It is important to keep in mind that students worked through the same curriculum regardless of modality. In other words, students working in the block-based interface were given the same assignments as those working in text. This lack of a significant difference suggests that the increased enjoyment of programming found in other studies using block-based tools (e.g. Wilson & Moffat, 2010) may have more to do with the curriculum used or the context in which learners programmed than the modality itself.

#### *4.2.3. Interest in future CS*

The final attitudinal survey result we presented in this analysis asks about students' interest in pursuing future computer science learning opportunities. It asked students to give a response on a 10-point Likert scale to the prompt: I Plan on Taking More Computer Science Courses after this one. Student responses at all three points in time, grouped by condition are shown in Fig. 5.

Like with the content scores presented in Fig. 3, when looking into students' interest in future computer science courses, we see students in the two conditions start and end the study at the same point with a significant difference emerging at the midpoint of the study. The first five weeks of the study saw participants in the Blocks condition become more interested in computer science courses, while students in the Text condition became less interested. At the midpoint of the study, the two conditions have significantly different levels of interest in future computer science coursework ($U = 264.5$, $p < .05$). After 10 weeks of working in Java, the trend flips with students coming from the Blocks environment becoming less interested, while students coming from the Text condition become more interested.

The results of this analysis are particularly interesting as they have potential implications for the strategic timing of the blocks-to-text transition in formal contexts. If the designers of an introductory computer science course sequence plan to have students use a block-based environment initially with the goal of transitioning learners to text-based programming at some point, the data shown in Fig. 5 suggest there are better and worse times to schedule the transition. For example, if the course sequence is set to have the first course be all in blocks and the second course only use text-based languages (which is a common approach), this data suggests that there will be a high enrollment rate in the second course, but that students' interest will wane early on. This pattern matches reports of educators implementing this approach (Cliburn, 2008). The potential downfall of this approach is that if the text-based introduction happens at the beginning of the course, it is more likely students will transfer out or drop the course. Alternatives to this approach would be to make the transition within the course so the teacher has built up a rapport with students and employ explicit bridging techniques (Dann et al., 2012), or to interleave the block-based and text-based instruction throughout the year (Powers et al., 2007). It is also important to note that there are other reasons to use a blocks-first sequence despite this finding, such as the greater learning gains (as discussed in section 4.1) or the prioritization of exposure to foundational concepts and easy introduction over preparation for future learning and transition to text-based programming that is emphasized in this study.

### *4.3. Programming practices outcomes*

The final dimension of our operationalization of impact investigates differences in programming practices between students coming from block-based tools compared to those coming from a text-based introduction to programming. Unlike the prior two Findings sections that used surveys and written assessments, this section analyzes the programs learners wrote and how they wrote them to further identify potential impacts of introductory modality. This section looks at a few different dimensions of the practice of

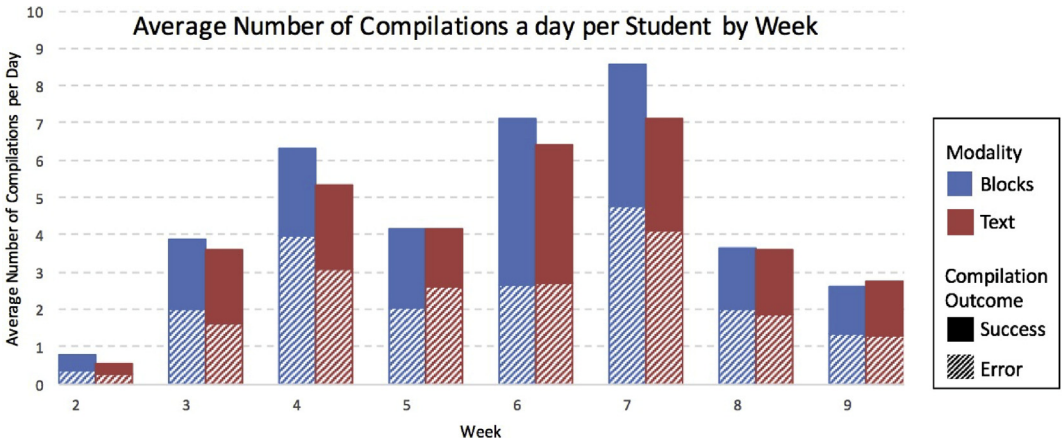


**Fig. 6.** The average number of `javac` calls per student per day grouped by week. The solid portion of each bar represents successful calls; the striped portions represent erroneous calls.

programming that have been investigated elsewhere in the computer science education literature to understand the process learning to program: compilation and run behaviors (Jadud, 2005; Jadud & Henriksen, 2009; Vihavainen, Luukkainen, & Ihantola, 2014, pp. 21–26), types and frequency of errors (Altadmri & Brown, 2015; Jackson, Cobb, & Carver, 2005; Jadud, 2005), and changes made between sequential runs (Berland, Martin, Benton, Petrick Smith, & Davis, 2013; Blikstein et al., 2014; Piech, Sahami, Koller, Cooper, & Blikstein, 2012). In this section, we use the data generated when student try and compile the Java program using the `javac` command as a means to understand emerging programming practices.

#### 4.3.1. *Differences in how often students attempted to run their programs*

Students in the Blocks condition ran the `javac` command an average of 142.3 times (*SD* 67.1) over the course of the ten weeks. Over that same period, students in the Text condition called `javac` 130.9 (*SD* 61.1) times, a difference that is not statistically significant ($t(56) = 0.67$, $p = .50$). In other words, there was no significant difference in the number of calls to `javac` based on the introductory modality students used. Fig. 6 shows the average number of compilations for each student per day across the three conditions broken down by week. This chart includes both successful compilations (solid portions of each bar) as well as calls that resulted in an error (the striped portion of each bar).

While the number of calls per week fluctuates over the course of the ten-weeks due to variations of the in-class activity, across the study we see little difference between the two conditions. Taken together, this shows students in the two conditions had similar programming practices in terms of how often they attempted to run their programs, next we look at success rates to see if there are differences in how error-prone these programs were.

#### 4.3.2. *Differences in success rates of compilations*

If a student makes a syntactic error in their program, a call to `javac` returns an error response informing them of the error. Over the ten-week curriculum, students who used the text-based introductory environment had an average of 70.26 errors (*SD* 38.2) while students coming from the block-based introductory condition averaged 75.7 total errors (*SD* 34.3). As a percentage of the total calls to `javac`, students in the Text condition had errors on 53.7% of `javac` calls, compared to 53.2% of calls for Blocks students, this difference in error rates is not statistically significant ($t(56) = 0.58$, $p = .57$). The striped portions of the columns in Fig. 6 show the average number of unsuccessful `javac` calls per week. If we break up this data to look at difference per student, we again see similar patterns across the two conditions. Table 1 provides an overview of the frequency of failed compilations per student as well as information about the number of errors per failed `javac` call broken down by condition.

Here again, we see little difference between the two conditions. This leads us to conclude that the five weeks spent in the text-based introductory environment did not better prepare learners for later programming in a professional text-based language with respect to authoring less error-prone programs. At the same time, it did not negatively impact learners' ability to write correct programs compared to having prior programming experience be situated in a block-based environment.

**Table 1**
High-level descriptive patterns of failing compilations and errors over the course of the 10 weeks.

| | Failed `javac` calls per student | Compilation errors per student | Compilation errors per failed `javac` call |
|---|---|---|---|
| Blocks | 75.11 | 165.78 | 2.23 |
| Text | 69.55 | 164.26 | 2.21 |

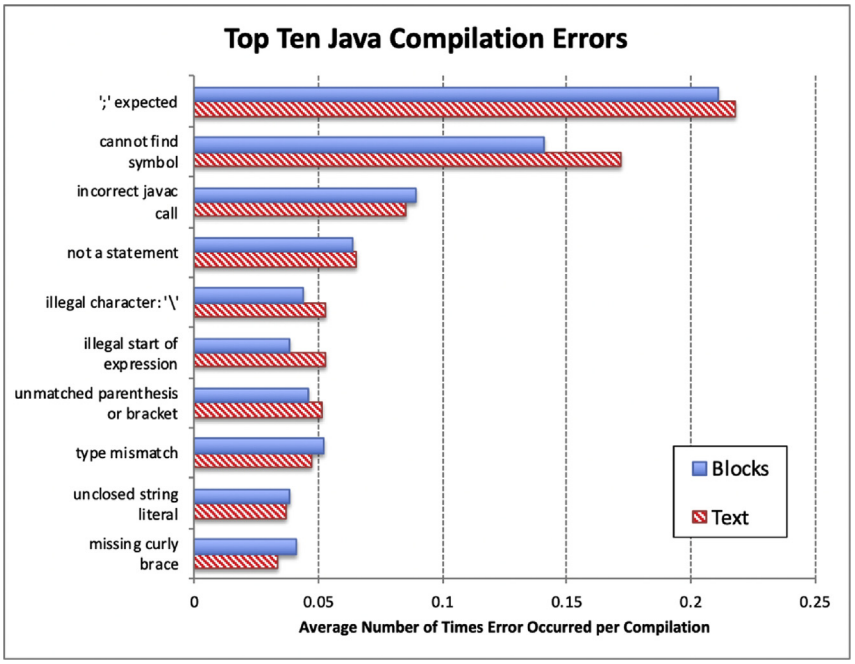


**Fig. 7.** The ten most frequently encountered Java errors, grouped by condition.

#### 4.3.3. Differences in the types of errors encountered

Just as we can glean information from patterns found in successful and erroneous compilation calls, we can also look at the types of programming errors the students encountered as another potential venue for differences between block-based and text-based introductions to emerge. This section reports on the types of errors encountered and their frequency to see if there are systematic differences based on learners' introductory modality. Unfortunately, the Java compiler often does not (and at times cannot) provide meaningful error messages to the programmer (or to researchers trying to understand novice programmer behaviors). For instance, a missing ';' could result in the error message "`expected ';' on line 11`" or by the rather generic message "`not a statement`". To make the analysis more meaningful, errors were grouped into more broadly defined error types, a detailed breakdown of how these error types were compiled can be found in (Weintrop, 2016). Fig. 7 shows the ten most frequently encountered errors grouped by condition. The values in this chart are reported on a per-compilation basis to control for differences in how often students compiled their programs.

The most common error was: "`';' expected`", which is seen when students forget to end a statement with a semicolon, a syntactic requirement of Java. The second most common error: "`cannot find symbol`", occurs when students try and use a variable before it has been defined. This pattern matches prior research looking at patterns of novice programming errors, as both Jadud (2005) and Flowers et al. (2004) identified these two mistakes as the most frequently encountered by novice programmers learning Java. Combined with the previous analysis, these data show that students in the two conditions not only made errors as the same rate but when they did have syntax errors in their programs, they were the same type of errors. The take away from this analysis is that introductory modality did not impact the type of errors learners made after transitioning to the Java programming language.

#### 4.3.4. Differences in the amount of code added in between successful compilations

Along with compilation patterns, we are also interested in how one goes about authoring a program. For example, does the learner write larger chunks of code and then see what happens? Or does the learner make small incremental edits en route to a functioning program? The idea is to investigate if introductory modality shaped the composition practice as it relates to the practice of authoring programs. To do this we look at the magnitude of changes made to programs between consecutive runs. To measure the size of the change, we use the Levenshtein distance between the texts of the two programs (Levenshtein, 1966). Levenshtein distance captures

**Table 2**
The frequency of successful compilations with a given Levenshtein distance from the last successful compilation of the same program.

| | Levenshtein Distance | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5–10 | 11–25 | 26–100 | > 100 |
| Blocks | 7.00 | 3.37 | 5.70 | 1.33 | 2.37 | 4.30 | 3.52 | 6.56 | 3.33 |
| Text | 6.16 | 3.00 | 5.58 | 1.23 | 2.13 | 3.77 | 3.48 | 5.87 | 2.55 |

the minimum number of single-character edits (i.e. insertions, deletions or substitutions) required to change one string into the other. Table 2 shows the results of this analysis. The columns capture the size of the Levenshtein distance between consecutive successful programs, while the cells show the average number of occurrences of that distance per student. The lower the number, the less often a program with that distance from the previous successful compilation was run by a student. For example, the left-most column that contains numbers shows that, on average, students in the Blocks condition compiled a program that was identical to the last program they compiled 7.00 times over the course of the 10 weeks, while the Text condition did this 6.16 times.

The students in the Text condition made fewer large changes to their programs and also re-ran their programs without making any changes less often than the other condition. These numbers tell the same story as the previous sections, namely that there is no systematic difference in the magnitude of program changes between consecutive runs between the two conditions after the transitions to Java. Also, it is worth noting, these data also allow us to reject the explanation that the fewer number of total complications in the Text condition was a result of the students making larger sets of changes between runs.

## 5. Discussion

The main contribution of this work is the finding that after seeing differences emerge between students working in block-based and text-based introductory environments, these difference fade after learners transition to a professional text-based environment. This lack of difference is true across conceptual, attitudinal, and programming practices dimensions. This is a significant finding as introductory computing curricula and environments designed for novices are becoming increasingly widespread so understanding the strengths (e.g. intuitive and accessible introduction to the field) and limitations (e.g. lack of immediate transfer of gained knowledge or practices) of the programming environment being used is important. One potential interpretation of these findings is that if the goal is to prepare learners to program in conventional text-based programming environments, then block-based tools may not be any more useful than the text-based introductory languages historically used. However, this interpretation ignores potential pedagogical and design approach that might facilitate a more successful transition to professional programming languages. We use these implications to shape our discussion section before concluding with some larger discussion points.

### 5.1. Implications for pedagogy

One of the study design decisions made for this work was to ask the teacher to not differentiate instruction between the two conditions. The intention was to try and control for pedagogy to be able to attribute differences in outcomes to the modality of the introductory tool. However, in doing so, there is a lost opportunity that could potentially explain the lack of transfer between the introductory tools and Java. In the block-based classroom, after the students moved to Java, the block-based environment was not seen again, nor was it referenced as a means to draw parallels between Java and the introductory tools that had been using for the first part of the course. In taking this approach, we limited the teacher's ability to use all the educational resources at her disposal. In post interviews with the teacher, she discussed how this study helped her see the potential of block-based programming in introductory programming courses and she commented on how in the future, she would continue to use the block-based environment, but interleave it throughout the year. She said in the future, she planned on introducing each new concept with the block-based tool before moving to Java, drawing direct parallels between the two. This pattern of a block-based introduction followed by a scaffolded text-based transition would then be repeated over the course of the year. While we do not have the data to report on the outcomes of this approach, the fact that this is what the teacher in this study decided would be the best course of action for her classrooms, speaks to the potential role of pedagogy in facilitating the transition between programming tools. Likewise, this also suggests a potential avenue of future research we hope to pursue.

### 5.2. Implications for the design of introductory programming environments

Just as pedagogy is one potential way to support learners in their transition from block-based to text-based programming, the programming environment itself may also facilitate this process. The programming environment used in this study (Pencil.cc) allowed learners to program in blocks or in text but did not allow students to move between the two. Pencil Code, the environment that Pencil.cc is built upon, was specifically designed to support this bi-directionality, allowing learners to move between block-based and text-based programming freely. Research looking at students working in these types of dual-modality environments shows how giving this control to the user can support learners with varying degrees of confidence as well as provide scaffolds to help learners author successful programs (Matsuzawa et al., 2015; Weintrop & Holbert, 2017). An alternative approach to dual-modality environments that support both blocks and text are hybrid environments that blend features of block-based and text-based tools into a single interface. For example, frame-based editors are an attempt to enable block-based-style editing but allowing users to mainly use the keyboard, rather than using the drag-and-drop interaction of most block-based tools (Kölling et al., 2017). Like with dual-modality tools, a growing body of research is showing that such environments are a promising approach for helping introduce novices to programming (Price et al., 2016; Weintrop & Wilensky, 2017c). The work presented in this paper suggests the need for such design solutions to help facilitate the transition from block-based to text-based tools by showing that gains made in block-based environments do not result in a head start in text-based tools.

### 5.3. Choosing an introductory programming environment

Given the growing interest in computing education, educators and parents are faced with the task of choosing an introductory programming environment to serve as a learner's first exposure to the world of programming. When choosing a programming environment (or modality for an environment) to be used in an education context, it is important to consider the question: what is the goal of this introductory computing experience? If the answer is: to prepare the learner for future computer science instruction, with the assumption that such a path will require learning to program in professional text-based tools, then the findings in this paper may cause the educator to reconsider if a block-based tool is the best approach. While the block-based introductory strategy did not impede learner progress when moving on to a professional language, neither did is facilitate it in a way that differed from an introductory text-based tool. Thus, if the educator has concerns related to block-based programming (e.g. the logistics of changing learning environments or learner concerns with the authenticity of block-based tools) or sees pedagogical utility in having learners type out programs from day one, it might be best to start learners with a text-based language rather than a block-based one.

However, if the goal of the learning experience is less about future computer science coursework and more focused on larger goals related to computational literacy and preparing learners to be informed citizens in a technological world, then a block-based tool may be a particularly effective introductory environment. In other words, deciding what the end goal of the introductory computing learning experience is should inform what modality to use. Likewise, this same logic applies when integrating programming is other disciplines or contexts. For example, if programming is being included in a science course to help develop mechanistic reason of a specific scientific phenomenon, then a block-based interface may be appropriate. Alternatively, if the goal is to prepare a student to conduct scientific research using existing scientific programming libraries written in a text-based language, then a text-based language might be preferable. Collectively, this work shows the importance of considering the goals and desired outcomes of introductory computing activities when choosing what programming modality to choose.

### 5.4. Implications beyond computer science educators and researchers

While the primary audience of this work will be those directly involved in the enterprise of helping young learners have positive and effective computer science learning experiences, there are potential takeaways for the larger audience of those interested in the role of computers in education. The main contribution of this work for larger audiences is highlighting the existence of an “expert blind spot” for those designing educational technologies. The notion of an “expert blind spot” is most often discussed with respect to teachers whose advanced knowledge of a subject results in them “not seeing” issues learners might have with a given concept (Nathan & Petrosino, 2003). In the case of this work, the people who are designing and implementing these introductory programming environments can see the connections between a block-based conditional statement and the textual equivalent so may assume learners will also see the connection. This research shows that this connection is not as clear to the learner as the designers may have hoped. The implication of this beyond the design of introductory programming environments is for all designers and researchers of educational technology to critically evaluate what knowledge might be taken for granted in their designs and highlight the role of empirical studies to explore potential expert blind spots.

A second, related, point of interest of this work for those outside of the computer science education community stems from the study design and the explanatory power that it enables. Given our interest in classroom practice and a desire to have results that speak to both researchers and educators, the study was specifically designed to be useful to both of those audiences. To accomplish this, care was taken to make the study setting as authentic as possible while also making the modification necessary to enable direct comparisons between experimental conditions. This meant embedding the study within existing classes and working with an experienced teacher. While this made the study more challenging to conduct, particularly in terms of finding a setting and getting the appropriate institutional approvals, the result was a dataset that holds up to the scrutiny of academics and teachers alike. If the goal of a research project is to shape classroom practice, being in classrooms is an important step towards achieving this goal.

A final contribution of this work is to demonstrate a multifaceted analysis of the potential roles of technology in education. The results section presented findings related to content learning, attitudinal outcomes, and practices associated with mastery of the content area. The study was designed to decouple these different dimensions of learning, meaning it was possible to see significant differences along any dimension independent of the findings of the other two. In doing so, there were multiple opportunities to find differences in outcomes between the two conditions. While no differences emerged, this study shows one possible way to study educational outcomes when learning is defined more broadly than just an increase in knowledge.

### 5.5. Limitations and future work

This study was designed to answer foundational research questions related to if and how introductory programming modality (block-based or text-based) impacts learners as they transition to professional text-based programming languages. While this study does provide insight into this question, there are limitations to the results. The first set of limitations of this study are related to the exceptionalities of the participants. The study took place in a selective enrollment school and in computer science classes where students had to sign-up to enroll. This means the study was comprised of learners that have historically been successful in formal educational contexts and showed a proclivity for computer science. Thus, the findings of this do not necessarily apply to all students, especially those with no interest in computer science. The ratio of male to female students is another limitation of the study as males were overrepresented. Finally, the teacher who participated in this study is an exceptional and experienced computer science educator. It is not clear how the results of this study would be different in a classroom led by a less experienced or less confident

instructor. While none of these limitations undermine the findings, they do limit the generalizability of this study and outline direction for future work. One clear direction forward would be a replication study at a different school with a different set of students. There are challenges associated with conducting such a study and recruiting teachers and students for it but it would be an important step forward for expanding the generalizability of this work.

A second limitation of the study is related to its 15-week duration. The ten weeks of the study was enough time to introduce the concepts but only a fraction of the year-long course. It is possible that differences between the conditions may emerge later in the semester or that different modalities impact the learning of some concepts that were not covered in the first 10 weeks of the course. While the study gave no reason to think this is the case, it is nonetheless possible that it takes longer than 10 weeks for the differences to emerge. A future iteration of this study lasting a full year could more fully explore this potentiality.

This study also has potential limitations stemming from the specific programming environments, languages, and curricula used. While Pencil.cc is a reasonable representative environment for both the larger category of block-based and text-based programming environments, in both cases it lacks some common features. For example, the block-based mode has a relatively constrained vocabulary and statement structure relative to tools like Scratch. At the same time, the text editor includes common scaffolds like syntax highlighting but does not include common features such as autocomplete or embedded code suggestions. Also, the decision to have CoffeeScript serve as the text-based programming language for the introductory portion of the study is only one of many possible language alternatives. Like with the participant-related limitations, conducting future work exploring the transition between different languages and different implementations of the block-based paradigm are open avenues of future work.

A related limitation stems from the transition from an introductory curriculum where students wrote procedural programs to an object-first Java curriculum. While this is a conceptual shift, we think this transition had a relatively small impact on the findings presented below for a few reasons. First, while the programs written in the Java portion of the study were object-oriented, intensive object-based content was rarely encountered. The programs authored focused more on basic input/output, the creation of variables (often using primitive types), and calling functions. When objects were encountered, the students were often given program templates to follow that included the code for object instantiation. A second reason to believe the impact was relatively limited is in how Pencil.cc handles sprites. When a new sprite is created, it is treated as an object. The command to create a new sprite is: `s = new Sprite()` and giving instructions to the new sprite uses dot notation (e.g. `s.fd 100`). As such, students encountered basic object-oriented concepts and syntax during the introductory portion of the course. Finally, as all students made the same transition, we expect any impact from the transition to be experienced by both conditions. All that being said, it is possible this shift in paradigms did have some impact on the results across all students, so is noted as a limitation.

Another limitation relates to methodological choices made on how we operationalized “impact” for this study. In focusing on conceptual understanding, attitudinal outcomes, and programming practices, we foreground only some of the many facets of learning to program. Absent from the analysis presented were measures such as program correctness, program quality, or overall ability to solve real problems through programming. While these measures are desirable, they were not a good fit for the context in which this work was conducted. First, concerning program correctness, the simplicity of the programs being authored and the level of scaffolding provided in the classroom. The teacher encouraged students to help each other and often worked alongside students during class time, especially those that were struggling. As a result, the final programs submitted for assignments were overwhelming correct for the assignment, thus making program correctness not a particularly useful measure in this specific context. In place of program correctness, we use characteristics of the process of authoring the program as a means to gain insight into the learners emerging understanding of concepts and development of programming skills. This allows us to see how the learner progressed over time rather than relying on the summative program. Evaluating program quality is difficult for similar reasons, stemming from the simplicity of the assignments and the complex and messy nature of classroom research. In terms of trying to evaluate learners’ abilities to solve real-world problems using programming, which is the ultimate goal of the class, after 10 weeks of learning Java in this course, the assignments students were being given were not at this level of sophistication. By the end of the study, students were still authoring relatively simple programs focused on introducing and applying basic concepts, rather than employing concepts to solve real-world problems. Later in the school year, programs shift to more applied and authentic challenges, but that happened well after the early work that was the focus of the first 10 weeks.

Along with this list of limitations, there are also other potential directions for future work that we hope to pursue. One possibility that was considered but not followed for this study is to investigate how modality differentially affects different types of students. For example, do students who are struggling with programming or are initially less interested in the discipline see more, less, or different benefits from one programming modality compared to another? Another direction of future work is taking a similar modality-centric approach to concepts beyond programming and computer science. As argued by Wilensky & Papert (2010), there is great promise and much work to be done in reimagining the representational infrastructure of a domain and the potential role of computationally mediated representational systems. As new fields become increasingly computational, new tools, interfaces, and modalities are being invented to support new computational endeavors. Research projects similar to this work could fit well amongst the various activities that accompany the formation of new technologically enhanced tools and practices and the larger emergence of new computational fields.

## 6. Conclusions

While block-based languages have exploded in popularity, relatively little research has been done to show that students learning in these environments are effectively transitioning emerging understandings and practices to more traditional text-based languages like Java (Blikstein, 2018). The goal of this work is to explore this transition in a high school classroom setting to understand if and

how modality facilitates this transition. Versions of this question have been answered in various ways in the literature (Armoni et al., 2015; Lewis, 2010; Price & Barnes, 2015; Saito et al., 2016; Weintrop, 2016), but never in a quasi-experimental setting where factors including teacher, curriculum, and time-on-task were controlled for and learners were from the same student body. The results of this high school classroom study show that students had greater learning gains in block-based environments compared to isomorphic text-based alternatives, but that these gains did not result in a difference in follow-on text-based instruction with respect to conceptual learning, attitudinal shifts, or successful programming practices.

By showing that conceptual gains made in block-based introductory tools do not automatically transfer to a professional text-based language, we shed a spotlight on the need for educators and tools designers to help facilitate this transition. Further, in identifying how modality shapes learners' attitudes towards the field of computer science, and how it changes as they shift programming languages, we can help support educators to make informed decisions about how best to welcome learners into the world of computing. As the role of computing and technology continue to grow in society, preparing young people to be informed participants in this technological landscape is important. Identifying the strengths and drawbacks of introductory programming tools that play a role in laying this foundational computational literacy is an important component of that process. The goal of this work is to move us closer to understanding how best to prepare all learners for the computational future that await them.

## References


Altadmri, A., & Brown, N. C. C. (2015). 37 million compilations: Investigating novice programming mistakes in large-scale student data. *Proceedings of the 46th ACM technical Symposium on computer science education - SIGCSE '15. Presented at the the 46th ACM technical Symposium* (pp. 522–527). Kansas City, Missouri, USA: ACM Press. https://doi.org/10.1145/2676723.2677258.

American Association of University Women (1994). *Shortchanging Girls, Shortchanging America.* Washington, DC: AAUW Educational Foundation.

Armoni, M., Meerbaum-Salant, O., & Ben-Ari, M. (2015). From Scratch to "real" programming. *ACM Trans. Comput. Educ. TOCE, 14*(25), 1–15.

Bart, A. C., Tibau, J., Kafura, D., Shaffer, C. A., & Tilevich, E. (2017). Design and evaluation of a block-based environment with a data science context. *IEEE Trans. Emerg. Top. Comput.* 1–1 https://doi.org/10.1109/TETC.2017.2729585.

Bau, D. (2015). Droplet, a blocks-based editor for text code. *J. Comput. Sci. Coll. 30*, 138–144.

Bau, D., Bau, D. A., Dawson, M., & Pickens, C. S. (2015). Pencil code: Block code for a text world. *Proceedings of the 14th International Conference on interaction design and Children* (pp. 445–448). New York, NY, USA: IDC '15. ACM. https://doi.org/10.1145/2771839.2771875.

Bau, D., Gray, J., Kelleher, C., Sheldon, J., & Turbak, F. (2017). Learnable programming: Blocks and beyond. *Communications of the ACM, 60*, 72–80. https://doi.org/10.1145/3015455.

Berland, M., Martin, T., Benton, T., Petrick Smith, C., & Davis, D. (2013). Using learning analytics to understand the learning pathways of novice programmers. *The Journal of the Learning Sciences, 22*, 564–599. https://doi.org/10.1080/10508406.2013.836655.

Bishop-Clark, C., Courte, J., & Howard, E. V. (2006). Programming in pairs with alice to improve confidence, enjoyment, and achievement. *Journal of Educational Computing Research, 34*, 213–228. https://doi.org/10.2190/CFKF-UGGC-JG1Q-7T40.

Blanchard, J. (2017). Hybrid environments: A bridge from blocks to text. *Proceedings of the 2017 ACM Conference on International computing education research, ICER '17* (pp. 295–296). New York, NY, USA: ACM. https://doi.org/10.1145/3105726.3105743.

Blikstein, P. (2018). *Pre-college computer science education: A survey of the field.* Mountain View, CA: Google LLC.

Blikstein, P., Worsley, M., Piech, C., Sahami, M., Cooper, S., & Koller, D. (2014). Programming pluralism: Using learning analytics to detect patterns in the learning of computer programming. *The Journal of the Learning Sciences, 23*, 561–599. https://doi.org/10.1080/10508406.2014.954750.

Brown, N. C. C., Mönig, J., Bau, A., & Weintrop, D. (2016). Future Directions of Block-based Programming. *Proceedings of the 47th ACM Technical Symposium on Computing Science Education* (pp. 315–316). New York, NY, USA: SIGCSE'16. ACM. https://doi.org/10.1145/2839509.2844661.

Bruckman, A., Biggers, M., Ericson, B., McKlin, T., Dimond, J., DiSalvo, B., et al. (2009). Georgia computes!: Improving the computing education pipeline. *ACM SIGCSE Bulletin*. ACM.

Caspersen, M. (2018). Teaching programming. In S. Sentance, E. Barendsen, & C. Schulte (Eds.). *Computer science education: Perspectives on teaching and learning* (pp. 109–130). Bloomsbury Publishing.

Cliburn, D. C. (2008). Student opinions of alice in CS1. *Frontiers in education Conference, 2008. FIE 2008. 38th Annual*IEEE T3B–1.

Code.org (2017). *The 5th Hour of Code is here! Code.org.* 9.20.18 https://medium.com/@codeorg/the-5th-hour-of-code-is-here-5b9ed3c29c50.

Code.org Curricula (2019). *Code.org*.

Cohen, L., Manion, L., & Morrison, K. (2007). *Research methods in education* (6th ed.). New York: Routledge, London .

Cooper, S., Dann, W., & Pausch, R. (2000). Alice: A 3-D tool for introductory programming concepts. *J. Comput. Sci. Coll. 15*, 107–116.

Cuny, J. (2015). Transforming K-12 computing education: An update and a call to action. *ACM Inroads, 6*, 54–57. https://doi.org/10.1145/2809795.

Danielak, B. A. (2014). *How electrical engineering students design computer programs.* College Park, MD: University of Maryland.

Dann, W., Cosgrove, D., Slater, D., Culyba, D., & Cooper, S. (2012). Mediated transfer: Alice 3 to Java. *Proceedings of the 43rd ACM technical Symposium on computer science education* (pp. 141–146). ACM.

Dorn, B., & Elliott Tew, A. (2015). Empirical validation and application of the computing attitudes survey. *Computer Science Education, 25*, 1–36. https://doi.org/10.1080/08993408.2015.1014142.

Duncan, C., Bell, T., & Tanimoto, S. (2014). Should your 8-year-old learn coding? *Proceedings of the 9th Workshop in primary and secondary computing education* (pp. 60–69). New York, NY, USA: WiPSCE '14. ACM. https://doi.org/10.1145/2670757.2670774.

Fay, M. P., & Proschan, M. A. (2010). Wilcoxon-mann-whitney or t-test? On assumptions for hypothesis tests and multiple interpretations of decision rules. *Statistics Surveys, 4*, 1–39. https://doi.org/10.1214/09-SS051.

Flowers, T., Carver, C. A., & Jackson, J. (2004). Empowering students and building confidence in novice programmers through Gauntlet. *Frontiers in education, 2004. FIE 2004. 34th Annual. Presented at the Frontiers in education, 2004. FIE 2004. 34th Annual: Vol. 1*, (pp. T3H/10–T3H/13). . https://doi.org/10.1109/FIE.2004.1408551.

Franklin, D., Skifstad, G., Rolock, R., Mehrotra, I., Ding, V., Hansen, A., Weintrop, D., & 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* (pp. 231–236). New York, NY, USA: SIGCSE '17. ACM. https://doi.org/10.1145/3017680.3017760.

Fraser, N. (2015). Ten things we've learned from Blockly. *2015 IEEE blocks and beyond Workshop (blocks and beyond). Presented at the 2015 IEEE blocks and beyond Workshop (blocks and beyond)* (pp. 49–50). . https://doi.org/10.1109/BLOCKS.2015.7369000.

Garcia, D., Harvey, B., & Barnes, T. (2015). The beauty and Joy of computing. *ACM Inroads, 6*, 71–79. https://doi.org/10.1145/2835184.

Garlick, R., & Cankaya, E. C. (2010). Using alice in CS1: A quantitative experiment. *Proceedings of the Fifteenth Annual Conference on Innovation and technology in computer science education* (pp. 165–168). ACM.

Good, J. (2018). Novice programming environments: Lowering the barriers, supporting the progression. *Innovative methods, user-Friendly tools, coding, and design approaches in people-oriented programming* (pp. 1–41). IGI Global.

Goode, J., Chapman, G., & Margolis, J. (2012). Beyond curriculum: The exploring computer science program. *ACM Inroads, 3*, 47–53.

Good, J., & Howland, K. (2017). programming Language, natural language? Supporting the diverse computational activities of novice programmers. *Journal of Visual*

*Languages & Computing, 39*, 78–92. https://doi.org/10.1016/j.jvlc.2016.10.008.
Grover, S., & 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* (pp. 267–272). New York, NY: ACM Press. https://doi.org/10.1145/3017680.3017723.
Grover, S., Pea, R., & Cooper, S. (2015). Designing for deeper learning in a blended computer science course for middle school students. *Computer Science Education, 25*, 199–237. https://doi.org/10.1080/08993408.2015.1033142.
Harvey, B. (1997). *Computer science logo style: Beyond programming.* The MIT Press.
Hill, C., Dwyer, H., Martinez, T., Harlow, D., & Franklin, D. (2015). Floors and Flexibility: Designing a programming environment for 4th-6th grade classrooms. *Proceedings of the 46th ACM technical Symposium on computer science education* (pp. 546–551). ACM.
Homer, M., & Noble, J. (2017). Lessons in combining block-based and textual programming. *J. Vis. Lang. Sentient Syst. 3*, 22–39. https://doi.org/10.18293/VLSS2017.
Horstmann, C. S. (2012). *Java concepts: Early objects* (7 edition). Hoboken, NJ: Wiley.
Howland, K., & Good, J. (2014). Learning to communicate computationally with flip: A bi-modal Programming Language for game creation. *Computers & Education*. https://doi.org/10.1016/j.compedu.2014.08.014.
Jackson, J., Cobb, M. J., & Carver, C. (2005). Identifying top Java errors for novice programmers. *Frontiers in education Conference*. IEEE.
Jadud, M. C. (2005). A first look at novice compilation behaviour using BlueJ. *Computer Science Education, 15*, 25–40.
Jadud, M. C., & Henriksen, P. (2009). Flexible, reusable tools for studying novice programmers. *Proceedings of the Fifth International Workshop on computing education research Workshop* (pp. 37–42). ACM.
Johnsgard, K., & McDonald, J. (2008). Using alice in overview courses to improve success rates in programming I. *IEEE 21st Conference on Software Engineering education and training, 2008* (pp. 129–136). CSEET '08. https://doi.org/10.1109/CSEET.2008.35.
Kelleher, C., & Pausch, R. (2007). Using storytelling to motivate programming. *Communications of the ACM, 50*, 58–64.
Kelleher, C., Pausch, R., & Kiesler, S. (2007). Storytelling alice motivates middle school girls to learn computer programming. *Proceedings of the SIGCHI Conference on Human factors in computing systems* (pp. 1455–1464). .
Kölling, M., Brown, N. C. C., & 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* (pp. 29–38). New York, NY, USA: WiPSCE '15. ACM. https://doi.org/10.1145/2818314.2818331.
Kölling, M., Brown, N. C. C., & Altadmri, A. (2017). Frame-based editing. *J. Vis. Lang. Sentient Syst. 3*, 40–67. https://doi.org/10.18293/VLSS2017.
Kölling, M., & McKay, F. (2016). Heuristic evaluation for novice programming systems. *Transactions on Computing Education, 16*, 12. 1–12:30 https://doi.org/10.1145/2872521.
Levenshtein, V. I. (1966). Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics* (pp. 707–710). .
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* (pp. 346–350). .
Malan, D. J., & Leitner, H. H. (2007). Scratch for budding computer scientists. *ACM SIGCSE Bulletin*. ACM.
Maloney, J. H., Peppler, K., Kafai, Y., Resnick, M., & Rusk, N. (2008). Programming by choice: Urban youth learning programming with Scratch. *ACM SIGCSE Bull, 40*, 367–371.
Maloney, J. H., Resnick, M., Rusk, N., Silverman, B., & Eastmond, E. (2010). The Scratch programming language and environment. *ACM Trans. Comput. Educ. TOCE, 10*, 16.
Margolis, J., & Fisher, A. (2003). *Unlocking the clubhouse: Women in computing.* The MIT Press.
Matsuzawa, Y., Ohata, T., Sugiura, M., & 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* (pp. 185–190). ACM Press. https://doi.org/10.1145/2676723.2677230.
McDowell, C., Werner, L., Bullock, H. E., & Fernald, J. (2006). Pair programming improves student retention, confidence, and program quality. *Communications of the ACM, 49*, 90–95. https://doi.org/10.1145/1145287.1145293.
Meerbaum-Salant, O., Armoni, M., & Ben-Ari, M. M. (2010). Learning computer science concepts with Scratch. *Proceedings of the sixth International Workshop on computing education research* (pp. 69–76). .
Mönig, J., Ohshima, Y., & Maloney, J. (2015). Blocks at your fingertips: Blurring the line between blocks and text in GP. *2015 IEEE blocks and beyond Workshop (blocks and beyond). Presented at the 2015 IEEE blocks and beyond Workshop (blocks and beyond)* (pp. 51–53). . https://doi.org/10.1109/BLOCKS.2015.7369001.
Nathan, M. J., & Petrosino, A. (2003). Expert blind spot among preservice teachers. *American Educational Research Journal, 40*, 905–928. https://doi.org/10.3102/00028312040004905.
Noone, M., & Mooney, A. (2018). Visual and textual programming languages: A systematic review of the literature. *Journal of Computers in Education. 5*, 149–174. https://doi.org/10.1007/s40692-018-0101-5.
Papert, S. (1980). *Mindstorms: Children, computers, and powerful ideas.* New York: Basic books.
Piech, C., Sahami, M., Koller, D., Cooper, S., & Blikstein, P. (2012). Modeling how students learn to program. *Proceedings of the 43rd ACM technical Symposium on computer science education* (pp. 153–160). ACM.
Powers, K., Ecott, S., & Hirshfield, L. M. (2007). Through the looking glass: Teaching CS0 with alice. *ACM SIGCSE Bull, 39*, 213–217.
Price, T. W., & Barnes, T. (2015). *Comparing textual and block interfaces in a novice programming environment. Presented at the ICER '15.* ACM Presshttps://doi.org/10.1145/2787622.2787712.
Price, T. W., Brown, N. C., Lipovac, D., Barnes, T., & Kölling, M. (2016). Evaluation of a frame-based programming editor. *Proceedings of the 2016 ACM Conference on International computing education research* (pp. 33–42). ACM.
Resnick, M., Silverman, B., Kafai, Y., Maloney, J., Monroy-Hernández, A., Rusk, N., et al. (2009). Scratch: Programming for all. *Communications of the ACM, 52*, 60.
Rodriguez Corral, J. M., Ruiz-Rube, I., Civit Balcells, A., Mota-Macias, J. M., Morgado-Estevez, A., & Dodero, J. M. (2019). A study on the suitability of visual languages for non-expert robot programmers. *IEEE Access, 7*, 17535–17550. https://doi.org/10.1109/ACCESS.2019.2895913.
Ruf, A., Mühling, A., & Hubwieser, P. (2014). *Scratch vs. Karel: Impact on learning outcomes and motivation.* ACM Presshttps://doi.org/10.1145/2670757.2670772.
Saito, D., Washizaki, H., & Fukazawa, Y. (2016). Analysis of the learning effects between text-based and visual-based beginner programming environments. *2016 IEEE 8th International Conference on Engineering education (ICEED). Presented at the 2016 IEEE 8th International Conference on Engineering education (ICEED)* (pp. 208–213). . https://doi.org/10.1109/ICEED.2016.7856073.
Scholtz, J., & Wiedenbeck, S. (1990). Learning second and subsequent programming languages: A problem of transfer. *International Journal of Human Computer Interaction, 2*, 51–72.
Shapiro, R. B., & Ahrens, M. (2016). Beyond blocks: Syntax and semantics. *Communications of the ACM, 59*, 39–41. https://doi.org/10.1145/2903751.
Simon, & Snowdon, S. (2014). Multiple-choice vs free-text code-explaining examination questions. *Presented at the Proceedings of the 14th Koli calling International Conference on computing education research* (pp. 91–97). ACM. https://doi.org/10.1145/2674683.2674701.
Streiner, D. L. (2003). Starting at the beginning: An introduction to coefficient Alpha and internal consistency. *Journal of Personality Assessment, 80*, 99–103. https://doi.org/10.1207/S15327752JPA8001_18.
Tabet, N., Gedawy, H., Alshikhabobakr, H., & Razak, S. (2016). From alice to Python. Introducing text-based programming in middle schools. *Proceedings of the 2016 ACM Conference on Innovation and technology in computer science education - ITiCSE '16. Presented at the the 2016 ACM Conference* (pp. 124–129). Arequipa, Peru: ACM Press. https://doi.org/10.1145/2899415.2899462.
Tangney, B., Oldham, E., Conneely, C., Barrett, S., & Lawlor, J. (2010). Pedagogy and processes for a computer programming outreach workshop—the bridge to college model. *Educ. IEEE Trans. On, 53*, 53–60.
Tavani, C. M., & Losh, S. C. (2003). *Motivation, self-confidence, and expectations as predictors of the academic performances among our high school students [WWW Document]. Child Study J.* 7.22.19 http://link.galegroup.com/apps/doc/A116924600/HRCA?sid=googlescholar.
Tempel, M. (2013). *Blocks programming. Vol. 9*. CSTA Voice.
Tew, A. E., Dorn, B., & Schneider, O. (2012). Toward a validated computing attitudes survey. *Proceedings of the Ninth Annual International Conference on International*

*computing education research* (pp. 135–142). ACM.
Vihavainen, A., Luukkainen, M., & Ihantola, P. (2014). *Analysis of source code snapshot granularity levels.* ACM Presshttps://doi.org/10.1145/2656450.2656473.
Weintrop, D. (2016). *Modality matters: Understanding the effects of programming language representation in high school computer science classrooms* (Ph.D. Dissertation) Evanston, IL: Northwestern University.
Weintrop, D., & 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* (pp. 633–638). New York, NY, USA: SIGCSE '17. ACM. https://doi.org/10.1145/3017680.3017707.
Weintrop, D., & Wilensky, U. (2018). How block-based, text-based, and hybrid block/text modalities shape novice programming practices. *Int. J. Child Comput. Interact. 17*, 83–92. https://doi.org/10.1016/j.ijcci.2018.04.005.
Weintrop, D., & Wilensky, U. (2017a). Comparing block-based and text-based programming in high school computer science classrooms. *ACM Trans. Comput. Educ. TOCE, 18*, 3. https://doi.org/10.1145/3089799.
Weintrop, D., & Wilensky, U. (2017b). How block-based languages support novices: A framework for categorizing block-based affordances. *J. Vis. Lang. Sentient Syst. 3*, 92–100. https://doi.org/10.18293/VLSS2017-006.
Weintrop, D., & Wilensky, U. (2017c). Between a block and a typeface: Designing and evaluating hybrid programming environments. *Proceedings of the 2017 conference on interaction design and children* (pp. 183–192). New York, NY, USA: IDC '17. ACM. https://doi.org/10.1145/3078072.3079715.
Weintrop, D., & Wilensky, U. (2015a). 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* (pp. 101–110). New York, NY, USA: ICER '15. ACM. https://doi.org/10.1145/2787622.2787721.
Weintrop, D., & Wilensky, U. (2015b). 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* (pp. 199–208). New York, NY, USA: IDC '15. ACM. https://doi.org/10.1145/2771839.2771860.
Weintrop, D. (2019). Block-based programming in computer science education. *Commun. ACM, 62*(8), 22–25. https://doi.org/10.1145/3341221.
Wiedenbeck, S. (1993). An analysis of novice programmers learning a second language. *Empirical studies of programmers: Fifth Workshop: Papers presented at the Fifth Workshop on empirical studies of programmers, December 3-5, 1993, Palo Alto, CA* (pp. 187). Intellect Books.
Wilensky, U., & Papert, S. (2010). Restructurations: Reformulating knowledge disciplines through new representational forms. In J. Clayson, & I. Kallas (Eds.). *Proceedings of the constructionism 2010 Conference. Paris, France*.
Wilson, A., & Moffat, D. C. (2010). Evaluating Scratch to introduce younger schoolchildren to programming. In: *Proc. 22nd Annu. Psychol. Program. Interest.* Group Univ. Carlos III Madr. Leganés Spain.