

Using Commutative Assessments to Compare Conceptual Understanding in Blocks-based and Text-based Programs

David Weintrop
Northwestern University
2120 Campus Drive, Suite 332
Evanston, Illinois 60628
dweintrop@u.northwestern.edu

Uri Wilensky
Northwestern University
2120 Campus Drive, Suite 337
Evanston, Illinois 60628
uri@northwestern.edu

ABSTRACT

Blocks-based programming environments are becoming increasingly common in introductory programming courses, but to date, little comparative work has been done to understand if and how this approach affects students' emerging understanding of fundamental programming concepts. In an effort to understand how tools like Scratch and Blockly differ from more conventional text-based introductory programming languages with respect to conceptual understanding, we developed a set of "commutative" assessments. Each multiple-choice question on the assessment includes a short program that can be displayed in either a blocks-based or text-based form. The set of potential answers for each question includes the correct answer along with choices informed by prior research on novice programming misconceptions. In this paper we introduce the Commutative Assessment, discuss the theoretical and practical motivations for the assessment, and present findings from a study that used the assessment. The study had 90 high school students take the assessment at three points over the course of the first ten weeks of an introduction to programming course, alternating the modality (blocks vs. text) for each question over the course of the three administrations of the assessment. Our analysis reveals differences on performance between blocks-based and text-based questions as well as differences in the frequency of misconceptions based on the modality. Future work, potential implications, and limitations of these findings are also discussed.

Categories and Subject Descriptors

D.1.7 [Visual Programming]. K.3.2 [Computer and Information Science Education]: Computer science education.

General Terms

Measurement, Design, Human Factors, Languages

Keywords

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

1. INTRODUCTION

A long-standing question faced by computer science educators is what language to use to introduce learners to programming. Ask this question to a room of ten teachers and you are likely to hear more than ten languages mentioned, many of which will carry

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

ICER '15, August 9-13, 2015, Omaha, NE, USA.

Copyright is held by the owner/author(s). Publication rights licensed to ACM. ACM 978-1-4503-3630-7/15/08...\$15.00.

DOI: <http://dx.doi.org/10.1145/2787622.2787721>

qualifiers describing under what conditions a given language is the best choice. These so called 'language wars' have been raging for as long as computer science has been taught, with little in the way of consensus emerging and with potentially detrimental effects [58]. Much work has been done attempting to empirically answer the question of which text-based language is best for novices, or at least identify features that make a language more or less accessible to beginners. While there is much to show for this effort, an alternative to conventional text-based languages is emerging in novice programming classrooms that brings a new dimension to introductory tools. Graphical blocks-based programming tools like Scratch [49], Blockly [23], and Alice [13] are becoming commonplace in introductory programming contexts, with a growing number of new curricula utilizing blocks-based programming tools in their materials, including the CS Principles project, the Exploring Computer Science program, and the materials being developed by code.org. The introduction of blocks-based programming environments changes the landscape of introductory tools, replacing questions of syntactic features of textual languages with the larger question of if text-based programming altogether is the best way to introduce novices to programming. Despite the increasing use of blocks-based tools in formal computer science learning contexts, relatively little work has investigated the cognitive affordances and drawbacks to the use of the graphical, blocks-based modality in classrooms. Similarly, few side-by-side studies have compared blocks-based and text-based tools directly (a notable exception being [32]). In their review of assessments of introductory programming, Gross and Powers [26] found that "one of the least studied questions are those that focus on how the environments impact a student's learning process and understanding from a formative perspective." In this paper, we set out to begin the process of filling in these gaps in the literature, specifically, we seek to answer the following two research questions:

1. How can we comparatively assess student understanding in blocks-based and text-based programming environments?
2. Does modality (blocks-based versus text-based) affect novice programmers' understanding of basic programming concepts? And if so, how does it differ by concept?

To answer to the first question, we created the Commutative Assessment, a novel programming assessment designed to measure students' understanding of programming concepts in both blocks-based and text-based modalities. Each question on the assessment requires the learner to read a short program (usually 4 or 5 lines) then answer a question about the outcome of the script. The key feature of the assessment is that every question can be asked with either a blocks-based or text-based program. In pursuit of our second question, the assessments were given three times over the course of a ten-week study in three introductory high

school programming courses. By administering the assessment at three points, students answered each question in both modalities. In this paper, we present the Commutative Assessment and share findings from its use as part of a larger study on the relationship between modality and student understanding.

2. Previous Research

2.1 Representations and Learning

“The tools we use have a profound (and devious!) influence on our thinking habits, and, therefore, on our thinking abilities.” [16]

As stated by the Turing Award winning computer scientist Edsger Dijkstra in the quote above, the tools we use, in this case the programming languages and development environments, have a profound, and often unforeseen, impact on how and what we think. diSessa [17] calls this material intelligence, arguing for close ties between the internal cognitive process and the external representations that support them: “we don’t always have ideas and then express them in the medium. We have ideas *with* the medium” [17 emphasis in the original]. He continues: “thinking in the presence of a medium that is manipulated to support your thought is simply different from unsupported thinking” [17]. The recognition that mental activity is mediated by tools and signs is one of the major contributions of the work of Vygotsky [65, 66] who argued that it is the external world that shapes internal cognitive functioning [72]. This perspective, coupled with Piaget’s constructivist learning theory, which contributes an interactionist perspective to learning that foregrounds the mutual dynamic of tools and thought [46], informs why it is so crucial to understand the relationship between the growing family of graphical programming representations and the understandings and practices they promote.

The role of representations in cognition has been studied across a variety of representational systems and their influence on various cognitive tasks. One large body of work that has emerged from studying this question is identifying the relationship between language, literacy and thought [33, 41, 66, 73], but as we are primarily concerned with the use of symbolic formalisms, we focus our review on scholarship looking at the role of arithmetic representation in supporting thought. The recognition that a learner’s own knowledge and experience influences the representations used and how it is understood and evaluated has been a recurring idea within the Learning Sciences [18, 31, 39, 52, 69]. For example, focusing on concepts from physics and investigating the use of conventional algebraic notation as compared to programmatic representations, Sherin [51] found that differing representational forms had different affordances with respect to students learning physics concepts and, as result, affected their conceptualization of the concepts learned. “Algebra physics trains students to seek out equilibria in the world. Programming encourages students to look for time-varying phenomena, and supports certain types of causal explanations, as well as the segmenting of the world into processes” [51]. Similar investigations have been done between programming languages. For example, Gilmore and Green [25] compared declarative and procedural notations and found that each notation afforded different types of reasoning. The procedural notation was superior for answering sequential questions while the declarative notation was better for answering circumstantial questions. This led them to conclude that “the structure of a notation affects the ease with which information can be extracted both from the printed page and from recall” [25].

Wilensky and Papert [74] use the term structuration to describe this relationship between the representational infrastructure used within a knowledge domain and the knowledge and understanding that the infrastructure enables and promotes. While often thought of as static, the structururations that underpin a discipline can change as new technologies and ideas emerge. Wilensky and Papert document a number of restructurations - shifts in representational infrastructure - including the move from Roman numerals to Hindu-Arabic numerals [61], the use of the Logo programming language to serve as a representational system to explore geometry [1], and the use of agent-based modeling to representation various biological, physical, and social systems [8, 50, 67, 75]. These shifts, and the new possibilities they enable, highlight the importance of studying representational systems, as restructurations can profoundly change the expressiveness, learnability, and communicability of ideas within a domain. While we are not claiming that the introduction of blocks-based tools constitutes a restructuration of programming knowledge, the recognition of the influence of representational infrastructure motivates this work and frames our larger program of research.

2.2 Programming Languages for Learners

Early on it was recognized that the design of a programming language itself can support or hinder students in their quest to master programming, which resulted in early efforts to develop more accessible programming languages [36]. Lead by Logo [22], which was explicitly designed with mathematics learning in mind, a number of languages emerged with the goal of serving as an introduction to the field of computer science. An early, influential language designed for novices was BASIC, whose acronym stands for Beginner’s All-purpose Symbolic Instruction Code. BASIC included a relatively small instruction set, removed all unnecessary syntax, and was designed to support short turn around times between composition and execution of programs, which collectively made it more accessible to novices.

As the field of computer science education matured, new languages and strategies emerged that were designed to serve as introductory tools and prepare learners for more powerful, fully featured languages. Languages such as Blue [30] and JJ [37] simplified syntax and provided tools to allow learners to focus on programming fundamentals before progressing to professional languages. Mini-languages, which are small languages designed to support the first steps in learning to program, are another approach for introductory languages [11]. These languages often center around specific activities and provided only the commands necessary to accomplish the immediate task, such as Karel the Robot, which has learners write short programs to control an on-screen robot [44]. Mini-languages are not intended for general purpose programming, they instead tailor the language around specific tasks, narrowing the gap between the objective and the representations in which intentions are encoded [15].

A final strategy that speaks directly to the work we are pursuing here is the creation of languages that try and address the documented issues that novices have with the syntax of programming languages. Research has found language syntax, the seemingly esoteric punctuation and formatting rules that must be followed when composing programs, can a serious barrier for novice programmers [14, 59]. Through a series of controlled experiments that had novices use one of a variety of languages that demonstrated various syntactic features, Stefik and Siebert [59] found that characteristics of syntax directly influence a language’s learnability. One solution to the problem of syntax is

the creation of visual programming tools that visually represent syntactic information of commands, making it easier to compose programs without encountering syntax errors.

2.3 Blocks-based Programming

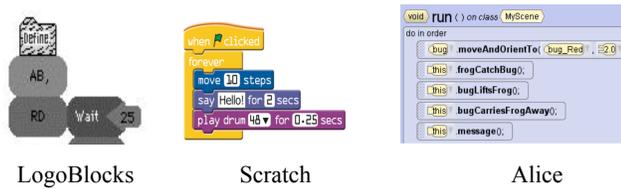


Figure 1. Three examples of blocks-based programming tools.

The blocks-based approach of visual programming (Figure 1), while not a recent innovation, has become widespread in recent years with the emergence of a new generation of tools, lead by the popularity of Scratch [49], Alice [13]Snap! [28], and Blockly [23]. These programming tools are a subset of the larger group of editors called *structured editors* [19] that make the atomic unit of composition a node in the abstract syntax tree (AST) of the program, as opposed to a smaller element (i.e. a character) or a larger element (like a fully formed functional unit). In making these AST elements the building blocks, then providing constraints to ensure nodes can only be added to the program’s AST in valid ways, the environment can prevent syntax errors. Blocks-based programming environments leverage a programming-primitive-as-puzzle-piece metaphor that provides visual cues to the user about how and where commands can be used as their means of constraining program composition. Programming in these environments takes the form of dragging blocks into a composition area and snapping them together to form scripts. If two blocks cannot be joined to form a valid syntactic statement, the environment prevents them from snapping together, thus helping to alleviate difficulties with syntax while retaining the practice of assembling programs instruction-by-instruction. This feature is especially relevant to the proposed study, as graphical programming proponents argue that visual depiction of syntax information is a key feature that contributes to its appropriateness for novice programmers [49]. However, other researchers are finding this approach does not solve the syntax problem, but merely delays it [43, 48]. Along with using block shape to denote usage, there are other visual cues to help novices, including color coding blocks by conceptual use, and nesting blocks to denote scope. Blocks-based programming has been found to be perceived as easier by learners, with a number of these visual features cited for its relative ease-of-use [70].

Early version of this interlocking approach include LogoBlocks [6] and BridgeTalk [9], which helped formulate the programming approach which has since grown to be used in dozens of applications. Alice [13], an influential and widely used environment in introductory programming classes, uses a very similar interface and has been the focus of much scholarship evaluating the merits of the approach at the undergraduate level. In addition to being used in more conventional computer science contexts, a growing number of environments have adopted the blocks-based programming approach to lower the barrier to programming across a variety of domains including mobile app development with MIT App Inventor and Pocket Code [53], modeling and simulation tools like StarLogo TNG [7], DeltaTick [76], NetFango [40] and EvoBuild [67], creative and artistic tools like Turtle Art and PicoBlocks, commercial educational tools like

Tynker and Hopscotch, game-based learning environments like RoboBuilder [68] and CodeSpells [21], and the activities included in Code.org’s Hour of Code, and Google’s Made with Code.

2.4 Programming assessments

Across educational research broadly there is a recognized need for high quality and validated assessments, a position echoed in computer science education circles [62]. Towards this end, a number of assessments have been developed and validated with the goal of improving our ability to evaluate and measure student learning across a variety of languages, environments, and contexts [27]. Related work has sought to define the process one follows to develop quality computer science assessments, beginning with identifying the goals of the assessment and the material to cover, through validating, piloting, and refining the instrument [12]. Additionally, new techniques are being developed and applied to programming assessments to improve accuracy and build confidence in new assessments [60]. One notable example of a rigorous, validated assessment is the Foundational CS1 assessment (FCS1) [64], which is a language independent instrument designed to decouple concepts from the language used to represent them. This makes it possible to be used with learners regardless of the language used during instruction. This is in contrast to most validated programming assessments developed by testing boards, like the Advanced Placement (AP) CS exam and the A-level General Certificate of Education in Computing, both of which are currently designed for the Java language.

There are a growing number of projects working towards developing assessments for the blocks-based approach to programming that we are investigating herein. Much of this work looks to assess not programming specifically, but computational thinking more broadly [27]. For example, the Fairy assessment [71], designed for middle school aged learners, uses Alice and presents learners with partially completed, or buggy, programs that need to be fixed in order for in-world characters to accomplish a specific task. In taking this approach, the Fairy assessment evaluates both comprehension (learners understanding of what a written program does) as well as gives learners a chance to problem solve, design and implement algorithmic solutions to assessment tasks. This design addresses the critique that process is often lost in conventional assessments of programming knowledge [47]. Another innovative assessment approach to computational thinking comes out of the Scalable Game Design group that developed an automated way to measure the frequency of computational thinking patterns in student-authored programs as a way to assess learning [29].

2.5 Evaluating Blocks-based Programming

A small, but growing body of literature is investigating the learning that happens with blocks-based programming tools. To date, most of this work has focused on Scratch and Alice, as these two environments have the widest use in contemporary computer science education. While both Alice and Scratch have been used in formal education environments, it is important to keep in mind that the two projects initially had different goals and targeted different age groups. Scratch from its inception, was focused on younger learners and informal settings [49], while Alice was targeted at more conventional computer science learning contexts and, as such, has a been the focus of more initiatives to evaluate student learning of programming concepts [13].

Ben-Ari and colleagues have conducted a number of studies on the use of Scratch for teaching computer science in formal

contexts [3, 4, 34, 35]. Meerbaum-Salant et al. [35] found that Scratch could successfully be used to introduce learners to central computer science concepts including variables, conditional and iterative logic, and concurrency. While students did perform well on the post-test evaluation in this project, a closer look at the programming practices learners developed while working in Scratch gave pause to the excitement around the results. The researchers found that students developed some undesirable programming habits, including a totally bottom-up programming approach, a tendency towards extremely fine-grained programming, and often unconventional, non-optimal usages of programming structures [34]. In a continuation of this study, the researchers concluded that students who learned Scratch in middle school more quickly grasped concepts in text-based languages when they reached high school (although they did not perform better on content assessments) [4]. Other work looking at comparing blocks-based to text-based programming using Scratch found that Scratch can be an effective way to introduce learners to programming concepts, although it is not universally more effective than comparable text languages [32]. There is also a growing body of work suggesting that the transition from blocks-based to text-based programming contexts is not as smooth as had once been assumed [24, 43, 48]. This suggests there are cognitive differences between these two programming modalities and is at the heart of the questions we pursuing here.

3. The Commutative Assessment

In pursuit of our first research question, we developed the Commutative Assessment as a way to evaluate if and how programming modality affects learnability. Each question on the assessment includes a short program for the student to read that can be expressed either in a blocks-based or text-based form. This means that no question relies on a construct unique to either modality, so for example, there are no questions that use blocks related to motion that students familiar with Scratch would recognize, as these instructions are not native to JavaScript. For each administration of the assessment, half of the questions are presented with blocks-based code and the other half use the text-based modality. The design of the Commutative Assessment makes it possible to group the responses along a number of dimensions that collectively yield insight into the relationship between modality and emerging understanding and provides data to support or refute claims about whether one modality is easier to interpret than another with respect to the various concepts.

To decide what concepts to include in our assessment, we primarily drew on two resources: the recently released 2013 CS Curriculum [2] and the work of Tew and Guzdial [63, 64]. In making the FCS1 assessment, Tew and Guzdial reviewed the contents of 12 introductory computer science textbooks along with other published curricula to establish a list of ten core CS1 concepts. Of this list, we chose to include five concepts in our assessment: fundamentals (variables, assignment, etc.), selection statements (conditional logic), definite loops (for loops), indefinite loops (while loops), and function/method parameters. Based on our review of the CS2013 Curriculum and what it emphasizes for introductory courses, we decided to add two additional content categories: program comprehension and algorithms (natural language descriptions of steps to be followed to solve a problem). As the algorithm questions do not include blocks-based or text-based programs, they are not discussed here.

The Commutative Assessment includes 28 questions, five each for conditional logic, loops, functions, and algorithms, and four

from the categories of variables and comprehension. While we would have liked to include a larger number of questions, we were constrained by the length of class and an awareness of testing fatigue effects from long assessments. All of the questions are multiple choice or true/false and, with the exception of the algorithm questions, take the form of a short piece of code that students are asked to interpret. The multiple choice answers were informed by misconceptions that have been identified in the literature (see appendix A of [56] for a summary of misconceptions). Figure 2 shows a sample variable question from the assessment. When taking the assessment, students see either the text version or the blocks version of the program.

What will be the value of x and y after this script is run?

```
var x = 10;
var y = x;
x = (x + 5);
```

vs.

A) x is equal to 15 and y is equal to 15
 B) x is equal to 5 and y is equal to 10
 C) x is equal to 15 and y is equal to 10
 D) x is equal to "x + 5" and y is equal to "x"
 E) x is equal to 10, 15 and y is equal to 10

Figure 2. A question from the Commutative Assessment.

The set of available choices includes the correct answer as well as responses drawn from the literature on misconceptions around variable assignment. Option A would be chosen by a student that holds the misconception that when one variable is assigned to another, the two values are linked and that whatever happens to one, happens to the other [10]. If a student incorrectly thinks that a value gets passed from one variable to another (i.e. the variable does not retain its value if another variable is set to it), then the student would choose option B. Option D would be chosen by a student who thinks expressions do not get evaluated during assignment [5, 55]. Finally, option E would be chosen by students who think that variables "remember" prior values [10, 20]. We also choose to write out "is equal to" instead of using an equals sign to be explicit about the meaning of the choices. Throughout the assessment we tried to follow this approach as much as possible to shed light on potential misconceptions conveyed or supported by the different modalities.

It is important to note that while the goal of this assessment is to understand the effect of programming modality on learning, there are other factors complicating the issue, most notably, differences in the language itself. For example, in Figure 2, the syntax and keywords used in variable declaration and assignment are different between the two modalities, making the difference between the two forms of the question more than just a shift in modality. This is a constant challenge with this work as a feature of the blocks-based modality is the ability to support more conversational and readable commands [70]. We will return to this challenge through the paper.

4. Methods and Participants

The data presented in this paper are part of a larger study comparing blocks-based, text-based, and hybrid blocks/text programming environments at a selective enrollment public high school in a Midwestern city. We followed students in three sections of an elective introductory programming course for the first 10 weeks of the school year. Each class spent the first five weeks of the course working in a blocks-based programming

environment. The students then transition to Java for the next five weeks of the study and then continued with Java for the remainder of the year. Two teachers participated in this study (one teacher taught two of the classes), both of whom had over five years of computer science teaching experience at the high school level.

The Commutative Assessment was administered online during class time at three points over the course of the 10-week study: at the outset, at the midpoint (end of week 5), and the conclusion of the study (end of week 10). Each time students took the assessment, they were asked the same set of 28 questions but the order and the modality (blocks vs. text) changed between administrations. The questions on the second content assessment used the opposite modality from the first assessment, so after taking the content assessment twice, all students had seen every question in both modalities. For the third assessment, two versions of the assessment were created that asked question in the same order, but varied modality. Students were then randomly given one of the two versions of the third assessment.

For the first five weeks of the course, each class used a slightly different programming environment based on Snap! [15]. Snap! is a blocks-based programming tool that is very similar to Scratch, but adds a few features (notably Snap! has first-class functions), and is implemented in JavaScript. The first class used a version of Snap! that gave students the ability to right-click on any block or script to see a JavaScript implementation of the program (Figure 3). In this tool, students were able to read, but not edit or write, text-based versions of the programs they constructed with the blocks. The second class used a version of Snap! that allowed students to read their programs in text and added the ability to define the behavior of new custom blocks in JavaScript. This served as a hybrid blocks/text read/write environment, as students could both read a text-based version of their own blocks, as well as write the behaviors of new blocks in JavaScript. The final class served as a control and used an unmodified version of Snap! All three classes followed the same curriculum based on UC Berkeley’s Beauty and Joy of Computing course, which covers all concepts included in the Commutative Assessment.

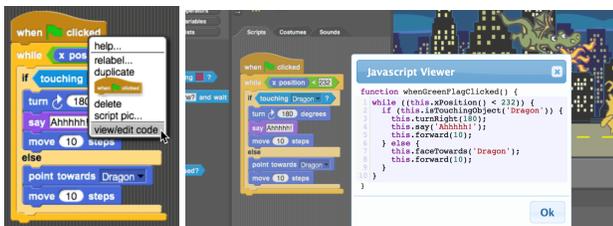


Figure 3. Side-by-side blocks and text in our version of Snap!

At the conclusion of the 5-week blocks-based introduction, the students transitioned to Java, following an objects-first curriculum. During the Java portion of the study, the topics covered in class included how to compile and run Java programs, simple data input and output, and the basics of defining and calling functions. While Java and JavaScript have syntactic differences, few of these differences were encountered by students during the five weeks of Java, the notable exception being the existence of variable types in Java as opposed to JavaScript’s weak typing. This difference was discussed by the teachers and was not identified as problematic by students during the study.

The school we worked with was chosen as it has a large computer science department, offering three sections of their Programming I course. A total of 90 students across three sections of the course participated in the study, which included 67 male students and 23 female students. The students participating in the study were 43% Hispanic, 29% White, 10% Asian, 6% African American, and 10% Multi-racial - a breakdown comparable to the larger student body. The classes included one student in eighth grade, three high school freshman, 43 sophomores, 18 juniors, and 25 high school seniors. Two-thirds of the students in the study speak a language other than English in their homes.

5. Results

As our research questions focus on the relationship between modality and concept, the first step of our analysis was to come up with a score for each concept/modality pair for every participant in the study. This means for each student we had 10 unique scores, one for each concept/modality tuple (variable/text; variables/blocks; loops/text, loops/blocks, etc.), resulting in 180 data points for each concept (90 students * 2 modalities). These scores were calculated by averaging together the student’s score for every question that fell into the tuple. Grouping this way helps us control for features of specific questions, and gives us a more accurate within-participant score for conceptual understanding by modality. These scores were then aggregated across the full set of participants to determine the relationship between concept and modality. We do not present a breakdown of responses by condition or time period. As this is our first analysis of data from the Commutative Assessment, we chose to focus on general outcomes, specifically looking for patterns and differences in student responses by concept/modality. Figure 4 shows the difference found for each concept.

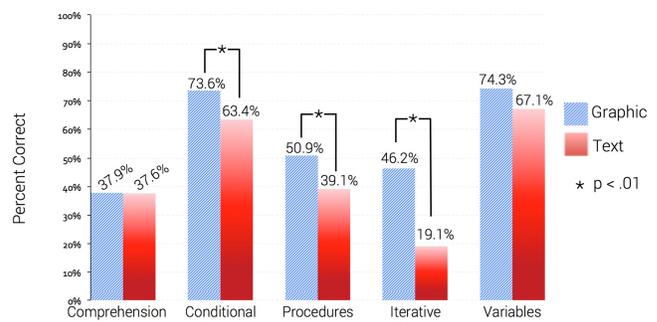


Figure 4. Student performance on the Commutative Assessment grouped by modality and concept.

Looking across the five conceptual categories covered in the Commutative Assessment using paired-samples t-tests shows that students in the graphical condition perform significantly better with the blocks-based modality on questions related to iterative logic $t(178) = 10.40, p < .001, d = 1.57$, conditional logic $t(178) = 2.82, p < .01, d = .41$ and functions $t(178) = 2.89, p < .01, d = .41$. Students also performed better in the graphical condition on variable questions, but not significantly so, $t(178) = 1.66, p = .10, d = .25$. Interestingly, there was almost no difference in how students performed on the comprehension questions between the two modalities $t(178) = .094, p = .92, d = .01$. These data suggest that the answer to the first part of our second research question is yes, modality does affect novice programmers’ understanding of basic programming concepts. Further, these data show that the effect is not uniform across concepts and does not seem to

influence comprehension of programs in the same way it effects basic understanding of what a construct does within a program. Seeing that a difference does exist, we now further investigate each category to answer the second part of our second research question, looking at how specific concepts are differentially influenced by modality and if they can be explained by misconceptions from the literature.

5.1 Iterative Logic Questions

While iterative logic showed the largest difference in scores between blocks-based and text-based questions, a closer analysis of the questions shows that a majority of this difference can be attributed to the difficulty students have with the structure of `for` loops [10]. Two of our five iterative logic questions compared a graphical `repeat` block to a text-based `for` loop (Figure 5).

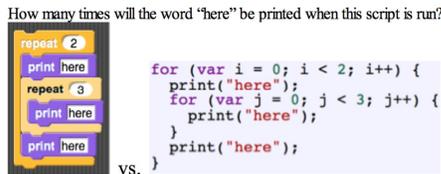


Figure 5. A sample iterative logic question.

On these two questions, students performed significantly better in the graphical condition (83% correct) versus the text-based `for` loop version of the question (16.1% correct). This provides compelling evidence for the finding that students find the `repeat` command common to blocks-based languages easier to understand than text-based `for` loops, a finding already documented in the literature [57, 59]. By examining the incorrect responses given by students, we can glean additional information about how students understand the concepts with respect to the way they are presented. For example, on the text-based `for` loop questions, almost half of the students (49.3%) chose an answer that had each command inside the `for` loop run once and only once – suggesting it was not clear that any looping was going to occur. When answering the same questions with the graphical `repeat` blocks, only 1.5% of students chose those options. Second, in the text-based conditions, 20.7% of students chose the answer that suggested the number of times a given for loop would run was variable, and would be different each time it was executed. In the graphical `repeat` versions of the questions, only one student chose this option. The Commutative Assessment includes one looping question that compared a blocks-based version of a `for` loop to a text-based version (Figure 6).

How many times will the comparison `c > 5` be tested when this program is run?

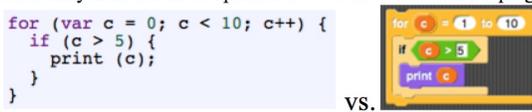


Figure 6. Comparing blocks-based and text-based for loops.

On this question, students performed comparably, answering the question correctly 19.6% percent of the time in the graphical condition and 18.0% of the time in the text-based condition. One possible explanation for the lack of difference on this question compared to what we saw on the two questions that use `repeat` is the confusion around the use of the term “for” to capture the concept of looping and the lack of transparency in how `for` loops behave based on this conventional representation [10, 57]. This outcome, along with the other for loop questions adds to the

evidence that students find the word “for” unintuitive, and that “repeat” better describes the looping behavior. As there are languages that utilize the keyword “repeat” (Logo in particular comes to mind), this finding speaks more to language design than features of the modality.

The two indefinite loop questions use the `while` construct. There was little difference in performance between the blocks-based and text-based versions of these questions. For both questions, students’ performance was very similar (a difference of .6% and 2.3% for the two questions). A closer investigation of the answers given (include incorrect answers) does not show a systematic difference between the types of representations used. This suggests that on indefinite loops, the blocks-based representation does not seem to provide any distinct advantage over a comparable text-based implementation. The lack of a difference between the two modalities when using comparable syntax/keywords, both with `while` loops and `for` loops, matches the finding from Lewis [32], who found no significant difference in accuracy between questions asked using the `repeat` block in Scratch and the `repeat` command in Logo. This suggests that for iterative logic, the blocks-based representation does not provide additional conceptual support; meaning the nested scoping and visual syntactic information did not better support student comprehension. A closer analysis of the five iterative logic questions only reinforces what we already know about the difficulty learners have with `for` loop syntax.

5.2 Conditional Logic questions

Students performed significantly better in the blocks-based modality on three of the five conditional logic questions. On one question the students performed comparably (.34% better on the blocks-based form), and on the last question students performed slightly better on text, scoring only 2.72% higher. On this final question, students were asked about the overall behavior of the script, as opposed to just the output, making it closer to our comprehension questions than the others, which may in part explain the better performance for the text-based representation - we will return to this issue later in the paper. On the three questions where students performed better in the graphical condition, two patterns emerged in analyzing the incorrect responses, revealing a slight systematic bias. First, on the two questions where the test of an `if/else` statement evaluated to true, students in the text condition were more likely to think both the `if` and the `else` branches would execute (11.5% for text versus 7.1% in the graphical case). This misconception has been identified in the literature [54] and is part of the work showing the `if/else` construct to be challenging for learners. In the current version of the Commutative Assessment only one of our five questions exposes this misconception, so we cannot make a strong claim about this error being alleviated by the blocks-based representation, but we plan on addressing this shortcoming in the next iteration of the assessment. Second, we found that students in the text condition were more likely to think the last statement is the one that is evaluated regardless of the outcome of the conditional logic surrounding it. On all three questions where this was a possible incorrect answer, students were more likely to choose it in the text-based condition (10.7% for text, versus 3.5% in blocks). This could be explained a number of ways, including students thinking that the body of a conditional statement gets executed regardless of the outcome of the conditional test, thinking the `else` outcome is always evaluated (which matches the first misconception identified and could explain two of the

three questions we saw this error in), or not know how or when conditions evaluate to true so defaulting to falling through to the last statement. Overall, the finding that students performed better on blocks-based conditional logic questions matches Lewis' previous work [32].

5.3 Variables Questions

Like with the two previous conceptual categories, students performed better (although not at a statistically significant level) on the variable questions when they were presented in the blocks-based form. A more detailed look reveals that students only performed better on the graphical case on three of the four questions in this category. On the one question that students performed better in the textual modality (Figure 7), one difference stands out from the others: variables are set then used, but never re-assigned, making it the simplest of the four questions.

What will be printed after running this script?

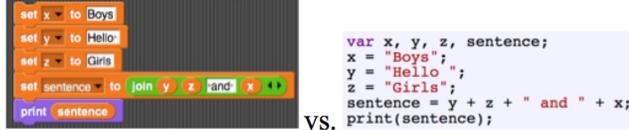


Figure 7. The variable question that students performed better in the text condition than the blocks-based condition.

This suggests that the text-based representation is comparable to the blocks-based version for simple variable assignment and usage, but that as statements and programs get more sophisticated (i.e. variables are assigned to other variables or variable values are set then reset), that the blocks-based modality is more intuitive for learners. As this is only a single question, we only mention it as a potential finding and plan to further investigate this in the future.

Looking at the incorrect responses given by students across the four variable questions reveals three findings that link modality to the existing misconceptions literature on variables. First, all four questions included an option that would be chosen by students who mistakenly thought expressions do not get evaluated as part of assignment (option D in Figure 2) and for all four questions, this incorrect option was chosen slightly more often in text form (7.3% of text responses, 5.3% of graphical). This could potentially be explained by the text form not providing visual hints about how to parse the statement. Second, we found that on text-based questions, students were more likely to incorrectly choose the answer that would result if variables held their initial values, meaning the values do not get overwritten (30.6% in text, 14.5% in graphical). We have not previously encountered this misconception in the literature. Our hypothesis is that in the case where students do not know what is supposed to happen when a variable that already contains a value has a new value set to it, the assumed behavior is for nothing to happen, i.e. the new value is ignored and the original value retained. Finally, students were also slightly more likely to choose answers that fit with the linked variables misconception (option A in figure 2) in the text questions (23.4% of text responses, 17.4% of graphical).

5.4 Function Questions

The fourth category of questions asked students about the outcome of running programs that contained function calls (Figure 8). On these questions, students performed better on the blocks-based version on four of the five questions we asked. Looking at the errors students made, we see a few cases where

students show signs of displaying documented misconceptions and other patterns that seem systematic, but are new to this work and can, at least partially, be explained by features of the modality. First, in one of our questions, we intentionally wrote a program that would output

Here are two functions:

```
function func1() {
  print("func1");
}
function func2() {
  print("func2");
  func1();
}
```

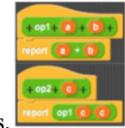
What is printed when this script is run?

```
func1
func2
vs.
func1()
func2();
```

(a)

Here are three functions that each perform a mathematical operation:

```
function op1(a, b) {
  return a + b;
}
function op2(c) {
  return op1(c, c);
}
```



What is output of this program?

```
print(op2(10)); vs.
report op1 c c
```

(b)

Figure 8. Two sample function questions.

the same word twice in a row, meaning the correct answer included the duplicated word while other choices included what students might assume was intended. Over half of the students (57%) in the text version of the question incorrectly chose the non-duplicated responses, compared to 38.6% of responses in the blocks-based version of the question. This suggests students found it easier to trace the flow in the blocks-based modality and were less likely to fall victim to what Pea [45] calls an "intentionality bug", where the learner assumes the computer knows the programmer's intention. A second systematic finding from analyzing these questions reinforces a trend observed in the variables questions, that students answering text-based questions were more likely to think that expressions do not get evaluated, but instead retain the expanded form (44% for text versus 31% of graphical responses). A third trend we found is that students were twice as likely (50% compared to 22%) to think that an unbounded recursive function stopped after a fixed number of calls in the text-based form than the blocks-based modality. Finally, two of our questions included functions that return values (report is the keyword used in the graphical form). Figure 8b provides an example of this type of question. Across these two questions, students were almost twice as likely to think the return command would cause an error in the text-based form (24.5% of responses) than the blocks-based alternative (13.2% of responses). In this case, we can point to a feature of the blocks-based modality that can account for this difference. In the blocks-based language, functions that return values are depicted as ovals or hexagons that need to be nested inside another block (like op2 in Figure 8b), whereas functions that do not have return statements take the shape of the interlocking blocks (like the func1 block in figure 8a). This visual difference at the place where the function is being invoked, and the ability for the blocks-based representation to enforce syntactic validity, provide a pair of scaffolds for the learner that potentially explains this difference in student responses in the two modalities.

5.5 Comprehension Questions

The final type of question on the assessment is program comprehension. These questions, unlike the others, focus more on what the purpose of a script is, as opposed to specific outcomes. In each case, the question students must answer is: what does the following script do? These questions require students to mentally run the program, often for different sets of potential inputs, and then interpret that behavior into a natural language description of

the behavior. Figure 9 shows two examples of these questions, with the correct answer being that the program swaps two values (left) and returns the largest of the three numbers (right).

Across the full set of questions, students performed comparably on the comprehension questions by modality (a difference of less than 1%). Looking at the questions individually, we see outcomes

a, b and tmp are variables. What does this script do? The function op4 takes in 3 numbers. What does op4 function do?

```
set tmp to a
set a to b
set b to tmp
```

VS.

```
tmp = a;
a = b;
b = tmp;
```

(a)

```
function op4(a, b, c) {
  var tmp;
  if (a > b) {
    tmp = a;
  } else {
    tmp = b;
  }
  if (c > tmp) {
    tmp = c;
  }
  return tmp;
}
```

VS.

(b)

Figure 9. Two comprehension questions

that correlate with the trends of how students did on questions from the conceptual category of the constructs used in the question. So, for example, question b in Figure 9, involves conditional logic and we found students performed better on the graphical versions of the question. Conversely, on a comprehension question that included a while loop, students performed better in the text condition. Because these questions involve the additional step of interpreting the behavior of scripts and the intention of the author, it becomes more difficult to map incorrect responses to specific misconceptions from the literature. Additionally, the small difference in performance between blocks-based and text-based questions is also interesting as it is the only category for which this is true, which leads to some potentially interesting conclusions. Notably, this suggests that while the graphical representation supports students in understanding what a construct does (i.e. what the output from using it is), that support does not better facilitate learners in understanding how to use that construct.

6. Discussion

The first research question we posed was how to comparatively assess understandings in two different modalities as part of the larger goal of studying the relationship between programming modality and understandings. The Commutative Assessment is our answer to that question. This assessment gives us the ability to directly compare responses to questions based on modality and concept and by giving the assessment at multiple time points, we are able to do both within and across student analyses of responses. Additionally, by providing responses based on misconceptions in the literature, we can link representational features of modalities with understandings that novices hold.

On three of our four conceptual categories we found significant differences in performance between modality, with the fourth category showing a similar, though less pronounced, trend. Three features of the blocks-based modality in particular stand out as possible explanations for this result. First, the graphical nesting of the blocks to denote scope appears to be an effective way to depict this concept, as we saw fewer errors made on blocks-based versions of questions where such misconceptions might be found. For example, students incorrectly thinking both branches of an if/else statement will be run was more prevalent in the text-based condition. The difference between {}s and visually nested commands provides one plausible explanation for this. Second,

the fact that the blocks-based modality allows for statements that can be closer to natural language can, in part, explain some of the differences we found. Notably, the command to assign values to variables takes the form of set __ to __, which is a closer description to what the command does than the comparable text-based language command of var __ = __. This difference is not a feature of the blocks-based modality, but instead an example of the language designer taking advantage of the more conversational format that the block-based modality enables. This difference can explain at least part of the differences we saw in the variable questions. Finally, the different shape of commands that return values from those that carry out actions in the blocks-based modality provides a compelling explanation for some of the differences we found in the function questions.

One of the more interesting outcomes from this work is the lack of difference between student performance on the comprehension questions. There are a few possible ways to explain this. One explanation is that the gains learners get from the graphical affordances of the blocks-based modality that support conceptual understanding of specific constructs does not carry over to slightly more challenging comprehension tasks. A second possible explanation is that it takes longer than the time allotted in the study for the gains from the graphical layout to apply to these types of questions. If this were the case, we would expect that if given more time, we would see similar gaps in performance emerge. A third possible explanation is that the modality has little effect on student comprehension. Although prior research would suggest otherwise, we continue to test this possibility. Teasing out which of these explanations is most accurate, or developing a potentially new explanation for this outcome is one direction this work is heading.

While we think the Commutative Assessment is a productive approach and can shed some light on the stated research questions, it is important to note what is not assessed by this work – the composition of programs. As such, the work we presented above only begins to answer our second research question on the relationship between modality and understanding. To more fully understand the relationship, additional data and complementary methods need to be applied. As part of this study we also conducted semi-structured clinical interviews with student and gathered log data of student programs. Our next step for this project is to use those data to triangulate patterns and relationships between the modalities and their cognitive affordances that we identified here. Additionally, the analyses presented herein did not account for time period or by Snap! condition. These are two dimensions we will pursue in future work. Finally, as previously mentioned, on some questions in the current form of the Commutative Assessment there is a conflation of modality and language features. While it is difficult to completely disentangle these characteristics of a programming language, in our next iteration of this study, we intend on using an environment where the language used in the blocks-based and text-based interfaces is syntactically more similar and uses a shared set of keywords and update the assessment with images from the new environment.

7. Conclusion

With the increasing presence of blocks-based programming in both formal and informal educational computing contexts, it is becoming increasingly important for us as educators and designers to more fully understand the effects of this modality on learners' conceptual understanding. The Commutative

Assessment allows us to systematically compare student understanding of fundamental concepts in blocks-based and text-based modalities, which in turn can give us insight into how learners are making sense of concepts using different representational tools. Through analyzing student responses, both correct and incorrect, we are starting to learn how blocks-based languages influence learners' emerging understandings and identify how modality can elicit or suppress misconceptions. The next step is to apply these findings to design new environments that will prepare the next generation of learners for the computational futures that await them.

8. References

- [1] Abelson, H. and DiSessa, A.A. 1986. *Turtle geometry: The computer as a medium for exploring mathematics*. MIT Press.
- [2] ACM/IEEE-CS Joint Task Force on Computing Curricula 2013. *Computer Science Curricula 2013*. ACM Press and IEEE Computer Society Press.
- [3] Armoni, M. and Ben-Ari, M. 2010. *Computer Science Concepts in Scratch*.
- [4] Armoni, M., Meerbaum-Salant, O. and Ben-Ari, M. 2015. From Scratch to "Real" Programming. *ACM Transactions on Computing Education*. 14, 4 (2015), 25.
- [5] Bayman, P. and Mayer, R.E. 1983. A diagnosis of beginning programmers' misconceptions of BASIC programming statements. *Comm. of the ACM*. 26, 9 (1983), 677–679.
- [6] Begel, A. 1996. *LogoBlocks: A graphical programming language for interacting with the world*. Electrical Engineering and Computer Science Department. MIT.
- [7] Begel, A. and Klopfer, E. 2007. Starlogo TNG: An introduction to game development. *Journal of E-Learning*. (2007).
- [8] Blikstein, P. and Wilensky, U. 2009. An Atom is Known by the Company it Keeps: A Constructionist Learning Environment for Materials Science Using Agent-Based Modeling. *Int. Journal of Computers for Mathematical Learning*. 14, 2 (2009), 81–119.
- [9] Bonar, J. and Liffick, B.W. 1987. A visual programming language for novices. *Principles of Visual Programming Systems*. S.K. Chang, ed. Prentice-Hall, Inc.
- [10] Du Boulay, B. 1986. Some difficulties of learning to program. *Journal of Educational Computing Research*. 2, 1 (1986), 57–73.
- [11] Brusilovsky, P., Calabrese, E., Hvorecky, J., Kouchnirenko, A. and Miller, P. 1997. Mini-languages: a way to learn programming principles. *Education and Information Technologies*. 2, 1 (1997), 65–83.
- [12] Buffum, P.S., Lobene, E.V., Frankosky, M.H., Boyer, K.E., Wiebe, E.N. and Lester, J.C. 2015. A Practical Guide to Developing and Validating Computer Science Knowledge Assessments with Application to Middle School. (2015).
- [13] Cooper, S., Dann, W. and Pausch, R. 2000. Alice: a 3-D tool for introductory programming concepts. *Journal of Computing Sciences in Colleges*. 15, 5 (2000), 107–116.
- [14] Denny, P., Luxton-Reilly, A., Tempero, E. and Hendrickx, J. 2011. Understanding the syntax barrier for novices. *Proc. of the 16th Annual ITiCSE* (2011), 208–212.
- [15] Van Deursen, A., Klint, P. and Visser, J. 2000. Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*. 35, 6 (2000), 26–36.
- [16] Dijkstra, E.W. 1982. How do we tell truths that might hurt? *Selected Writings on Computing: A Personal Perspective*. Springer. 129–131.
- [17] diSessa, A.A. 2000. *Changing minds: Computers, learning, and literacy*. MIT Press.
- [18] diSessa, A.A., Hammer, D., Sherin, B.L. and Kolpakowski, T. 1991. Inventing graphing: Meta-representational expertise. *Journal of Mathematical Behavior*. 10, (1991), 117–160.
- [19] Donzeau-Gouge, V., Huet, G., Lang, B. and Kahn, G. 1984. Programming environments based on structured editors: The MENTOR experience. *Interactive Programming Environments*. McGraw Hill.
- [20] Doukakis, D., Grigoriadou, M. and Tsaganou, G. 2007. Understanding the programming variable concept with animated interactive analogies. *Proc. of the 8th HERCMA Conference* (2007).
- [21] Esper, S., Foster, S.R. and Griswold, W.G. 2013. CodeSpells: embodying the metaphor of wizardry for programming. *Proc. of the 18th annual ITiCSE conference* (2013), 249–254.
- [22] Feurzeig, W., Papert, S., Bloom, M., Grant, R. and Solomon, C. 1970. Programming-languages as a conceptual framework for teaching mathematics. *SIGCUE Outlook*. 4(1970), 13–17.
- [23] Fraser, N. 2013. *Blockly*. Google.
- [24] Garlick, R. and Cankaya, E.C. 2010. Using Alice in CS1: A quantitative experiment. *Proc. of the 15th Annual ITiCSE Conference* (2010), 165–168.
- [25] Gilmore, D.J. and Green, T.R.G. 1984. Comprehension and recall of miniature programs. *Int. Journal of Man-Machine Studies*. 21, 1 (1984), 31–48.
- [26] Gross, P. and Powers, K. 2005. Evaluating Assessments of Novice Programming Environments. *Proc. of the 1st Annual ICER Conference* (NY, USA, 2005), 99–110.
- [27] Grover, S., Cooper, S. and Pea, R. 2014. Assessing computational learning in K-12. (2014), 57–62.
- [28] Harvey, B. and Mönig, J. 2010. Bringing "no ceiling" to Scratch: Can one language serve kids and computer scientists? *Proc. of Constructionism 2010* (Paris, Fr.), 1–10.
- [29] Koh, K.H., Basawapatna, A., Nickerson, H. and Reppenng, A. 2014. Real Time Assessment of Computational Thinking. *Visual Languages and Human-Centric Computing*, 49–52.
- [30] Kölling, M. and Rosenberg, J. 1996. Blue—a language for teaching object-oriented programming. *ACM SIGCSE Bulletin* (1996), 190–194.
- [31] Lave, J. 1988. *Cognition in practice: Mind, mathematics, and culture in everyday life*. Cambridge Univ Press.
- [32] Lewis, C.M. 2010. How programming environment shapes perception, learning and goals: Logo vs. Scratch. *Proc. of the 41st Annual ACM SIGCSE Conference* (NY, 2010), 346–350.
- [33] Luria, A.R. 1982. *Language and cognition*. Winston ; Wiley, Washington, D.C. : New York ; Chichester :
- [34] Meerbaum-Salant, O., Armoni, M. and Ben-Ari, M. 2011. Habits of programming in Scratch. *Proc. of the 16th Annual ITiCSE Conference* (Darmstadt, Germany, 2011), 168–172.
- [35] Meerbaum-Salant, O., Armoni, M. and Ben-Ari, M.M. 2010. Learning computer science concepts with scratch. *Proc. of the 6th Annual ICER Conference* (2010), 69–76.
- [36] Mendelsohn, P., Green, T.R.G. and Brna, P. 1990. *Programming languages in education: The search for an easy start*. Academic Press London.
- [37] Motil, J. and Epstein, D. 1998. JJ: a Language Designed for Beginners.

- [38] Nemirovsky, R. 1994. On ways of symbolizing: The case of Laura and the velocity sign. *The Journal of Mathematical Behavior*. 13, 4 (1994), 389–422.
- [39] Noss, R. and Hoyles, C. 1996. *Windows on mathematical meanings: Learning cultures and computers*. Kluwer.
- [40] Olson, I.C. and Horn, M.S. 2011. Modeling on the table: agent-based modeling in elementary school with NetTango. *Proc. of the 10th Annual IDC Conference*. (2011), 189–192.
- [41] Ong, W. 1982. *Orality and Literacy: The technologizing of the world*. Routledge.
- [42] Palmer, S.E. 1978. Fundamental aspects of cognitive representation. *Cognition and categorization*. E. Rosch and B.B. Lloyd, eds. Lawrence Erlbaum Associates. 259–303.
- [43] Parsons, D. and Haden, P. 2007. Programming osmosis: Knowledge transfer from imperative to visual programming environments. *Proc. of The 12th Annual NACCQ Conference* (Hamilton, New Zealand, 2007), 209–215.
- [44] Pattis, R.E. 1981. *Karel the robot: a gentle introduction to the art of programming*. John Wiley & Sons, Inc.
- [45] Pea, R.D. 1986. Language-independent conceptual “bugs” in novice programming. *Journal of Educational Computing Research*. 2, 1 (1986), 25–36.
- [46] Piaget, J. 1952. *The origins of intelligence in children*. International Universities Press, Inc.
- [47] Piech, C., Sahami, M., Koller, D., Cooper, S. and Blikstein, P. 2012. Modeling how students learn to program. *Proc. of the 43rd ACM SIGCSE Conference* (2012), 153–160.
- [48] Powers, K., Ecott, S. and Hirshfield, L.M. 2007. Through the looking glass: teaching CS0 with Alice. *ACM SIGCSE Bulletin*. 39, 1 (2007), 213–217.
- [49] Resnick, M. et al. 2009. Scratch: Programming for all. *Comm. of the ACM*. 52, 11 (2009), 60.
- [50] Sengupta, P. and Wilensky, U. 2009. Learning Electricity with NIELS: Thinking with Electrons and Thinking in Levels. *Int. Journal of Computers for Mathematical Learning*. 14, 1 (2009), 21–50.
- [51] Sherin, B.L. 2001. A comparison of programming languages and algebraic notation as expressive languages for physics. *Int. Journal of Computers for Mathematical Learning*. 6, 1 (2001), 1–61.
- [52] Sherin, B.L. 2000. How students invent representations of motion: A genetic account. *The Journal of Mathematical Behavior*. 19, 4 (2000), 399–441.
- [53] Slany, W. 2014. Tinkering with Pocket Code, a Scratch-like programming app for your smartphone. *Proc. of Constructionism 2014* (Vienna, Austria, 2014).
- [54] Sleeman, D., Putnam, R.T., Baxter, J. and Kuspa, L. 1986. Pascal and high school students: A study of errors. *Journal of Educational Computing Research*. 2, 1 (1986), 5–23.
- [55] Sorva, J. 2008. The same but different students’ understandings of primitive and object variables. *Proc. of the 8th Annual ICER Conference* (2008), 5–15.
- [56] Sorva, J. 2012. *Visual Program Simulation in Introductory Programming Education*. Aalto University.
- [57] Stefik, A. and Gellenbeck, E. 2011. Empirical studies on programming language stimuli. *Software Quality Journal*. 19, 1 (2011), 65–99.
- [58] Stefik, A. and Hanenberg, S. 2014. The Programming Language Wars: Questions and Responsibilities for the Programming Language Community. *Proc. of the 2014 Int. Symposium on New Ideas, New Paradigms, and Reflections on Programming* (NY, USA, 2014), 283–299.
- [59] Stefik, A. and Siebert, S. 2013. An Empirical Investigation into Programming Language Syntax. *ACM Transactions on Computing Education*. 13, 4 (2013), 1–40.
- [60] Sudol, L.A. and Studer, C. 2010. Analyzing Test Items: Using Item Response Theory to Validate Assessments. *Proc. of the 41st ACM SIGCSE Conference* (NY, 2010), 436–440.
- [61] Swetz, F. 1989. *Capitalism and arithmetic: The new math of the 15th century*. Open Court.
- [62] Tew, A.E. and Dorn, B. 2013. The Case for Validated Tools in Computer Science Education Research. *Computer*. 46, 9 (2013), 60–66.
- [63] Tew, A.E. and Guzdial, M. 2010. Developing a validated assessment of fundamental CS1 concepts. *Proc. of the 41st Annual ACM SIGCSE Conference* (2010), 97–101.
- [64] Tew, A.E. and Guzdial, M. 2011. The FCS1: a language independent assessment of CS1 knowledge. *Proc. of the 42nd Annual ACM SIGCSE Conference* (2011), 111–116.
- [65] Vygotsky, L. 1978. *Mind in society: The development of higher psychological processes*. Harvard University Press.
- [66] Vygotsky, L. 1986. *Thought and language*. MIT Press.
- [67] Wagh, A. and Wilensky, U. 2012. Evolution in blocks: Building models of evolution using blocks. *Proc. of Constructionism 2012* (Athens, Gr, 2012).
- [68] Weintrop, D. and Wilensky, U. 2012. RoboBuilder: A program-to-play constructionist video game. *Proc. of Constructionism 2012* (Athens, Gr, 2012).
- [69] Weintrop, D. and Wilensky, U. 2014. Situating programming abstractions in a constructionist video game. *Proc. of Constructionism 2014* (Vienna, Au, 2014).
- [70] Weintrop, D. and Wilensky, U. 2015. To Block or not to Block, That is the Question: Students’ Perceptions of Blocks-based Programming. *Proc. of the 14th Annual IDC Conference* (Boston, MA, 2015).
- [71] Werner, L., Denner, J., Campe, S. and Kawamoto, D.C. 2012. The fairy performance assessment: measuring computational thinking in middle school. *Proc. of the 43rd Annual ACM SIGCSE Conference* (2012), 215–220.
- [72] Wertsch, J.V. 1991. *Voices of the mind: A sociocultural approach to mediated action*. Harvard University Press.
- [73] Whorf, B.L., Carroll, J.B. and Chase, S. 1956. *Language, thought, and reality: Selected writings of Benjamin Lee Whorf*. MIT press Cambridge, MA.
- [74] Wilensky, U. and Papert, S. 2010. Restructurations: Reformulating knowledge disciplines through new representational forms. *Proc. of Constructionism 2010* (Paris, Fr., 2010).
- [75] Wilensky, U. and Reisman, K. 2006. Thinking like a wolf, a sheep, or a firefly: Learning biology through constructing and testing computational theories— an embodied modeling approach. *Cognition and Instruction*. 24, 2 (2006), 171–209.
- [76] Wilkerson-Jerde, M.H. and Wilensky, U. 2010. Restructuring Change, Interpreting Changes: The DeltaTick Modeling and Analysis Toolkit. *Proc. of Constructionism 2010* (Paris, Fr, 2010).