Understanding the Effects of Programming Representations in Introductory Environments: Comparing Text-based, Blocks-based, and Hybrid Blocks/Text Programming Tools[1]

David Weintrop
PhD Thesis Proposal
Department of the Learning Sciences
Northwestern University

[1] This a working title and is not yet finalized. Suggestions welcome.

Table of Contents

Summary

A long-standing questioned faced by educators when introducing learners to programming is deciding where to start on day one: What programming language to choose? In what environment? What activity should learners engage in? An increasingly popular approach to the design of introductory programming tools is the use of graphical, blocks-based programming environments that leverage a primitives-as-puzzle-pieces metaphor. In such environments, learners can assemble functioning programs using only a mouse by snapping together instructions and receive visual feedback on how and where commands can be used and if a given construction is valid. The use of this programming modality has become a common feature of introductory computer science curricula and programming interventions targeted at young learners. Despite its growing popularity, open questions remain surrounding the effectiveness of this representational system for helping students learn basic programming concepts and the overall suitability of the approach for preparing learners for future computer science learning opportunities. Further, it is unclear what the strengths and weaknesses of block-based programming tools are compared to isomorphic text-based alternatives, and relatively little work has been done investigating the design space between blocks-based and text-based programming tools. The goal of this dissertation is to shed light on these open questions and use the findings from investigating these questions to inform the design of a new, hybrid programming environment that draws on the strengths of the blocks-based programming approach while addressing identified drawbacks of the approach.

This dissertation seeks to answer three sets of research questions. The first set of questions pertain to the effects of programming modalities on students' resulting understandings of programming concepts and the programming practices they develop. There is a growing body of literature on the effects, both positive and negative, of using blocks-based programming environments with programming novices in formal education environments (e.g. Cooper et al., 2000; Lewis, 2010; Meerbaum-Salant et al., 2010, 2011). The first goal of this dissertation is to contribute to this literature by conducting in-depth evaluations of learners working with blocks-based, text-based, and hybrid blocks-text environments to identify how emerging understandings are influenced by the programming modality that learners use. The emphasis will be on understanding the role of the programming representations on students' emerging understandings of core programming concepts, the programming practices that develop, and students' overall perceptions of and attitudes towards programming.

The second set of questions look at the effectiveness of introductory programming tools for preparing students for future computer science learning opportunities. While blocks-based programming environments have been found to be successful at engaging students in programming activities and providing early success with little or no formal instruction (e.g. Maloney et al., 2008), educators have had difficulty transitioning learners from these graphical environments to conventional text-based programming environments. Studies have found little transfer in knowledge between graphical environments intended to introduce learners to programming and text-based environments that serve as the core modality for further computer

science studies (Chetty & Barlow-Jones, 2012; Garlick & Cankaya, 2010; Powers, Ecott, & Hirshfield, 2007). Students often fail to see the relationship between the two programming modalities, which calls into question the effectiveness of blocks-based graphical environments in their capacity to serve as an effective introduction to computer science (Cliburn, 2008). A small number of studies have focused on this problem, but have focused on undergraduate audiences and have relied on add-ons to programming environment, as opposed to being a part of the environment itself (Dann et al., 2012). This shows that bridging blocks and text is possible, but requires a dedicated effort built into the environment and curriculum.

In the first year of the study, we will investigate these two sets of questions, looking specifically at the blocks-based programming modality. Findings from year one will be used to answer our third set of research questions, which consists of design questions meant to guide the design of a hybrid blocks/text introductory programming tool. In the second year of the study, we will we will design, implement, and evaluate this tool alongside blocks-based and text-based alternatives as part of a 3-condition, quasi-experimental study in a yearlong high school programming course. The study will look at the effects of these three programming environment on learners' understandings of programming concepts, their attitudes towards and perceptions of computer science, and how well the different representations prepare learners for future computer science instruction in more conventional text-based programming environments. This study will use a mixed-method approach drawing on a variety of data sources, including clinical interviews, classroom observations, written pre/post assessments and surveys, teacher interviews, and the collection and analysis of student constructed artifacts.

As computer science instruction at the high-school level is becoming more widespread and the number of tools and curricula designed for the classroom grow, it is important that we are aware of the impact of using these tools may have, both positive and negative. This dissertation seeks to advance our understanding of the relationship between programming modality and resulting understandings of programming concepts, the practices that develop, and the attitudes toward and perceptions of programming they promote in learners. Further, this work will follow learners as they the transition from introductory programming environments to the tools used in later computer science educational contexts, in an effort to further assess the appropriateness of different representational systems used in introductory programming tools. Finally, as part of this dissertation a new, hybrid blocks/text programming environment will be developed and evaluated that will apply what is learned about the relationship between programming modality and student learning. In answering these questions and building these tools, we can be better equipped to provide advice and recommendations to computer science educators tasked with teaching future generations of programmers.

Introduction

Computation is changing our world. Nowhere is this more true than in the workplace, where entire industries are being upended and redefined as new technologies and computational tools emerge and replace the status quo. In this new landscape, competencies and skills grounded in the ability to effectively use computational tools, and design and implement solutions that rely on computation are highly valued. This skill set, often collected under umbrella terms like "Computational Thinking," or "21st Century Skills" has long been championed as critical for the next generation of learners to ensure they become productive contributors in our increasingly digital world. While few argue against the importance of preparing learners for the computational nature of the workforce, what exactly that entails, and how and when are the most effective ways to do so remain open questions. A recently renewed interest in bringing these skills into classrooms has produced a great deal of momentum towards bring computer science, in one form or another, into K-12 education. This has produced a number of large-scale initiatives that seek to teach the next generation of computationally literature citizens. Some of these approaches rely on students' self-motivation and provide online tutorials, videos, and a suite of programming tools and resources to lead learners through content. Others are trying to work with in formal educational settings to bring computer science into the mainstream. This undertaking includes a variety of efforts beyond the immediate challenges of designing curricula and assessments, these efforts include lobbying elected officials to change the categorization of computer science with respect to graduation requirements, partnering with school district administrators to adopt and supporting district-wide interventions, and large-scale teacher professional development opportunities to prepare teachers to teach these new materials.

Amidst all this excitement and activity around bringing computer science, and computational thinking more broadly, into schools and classrooms nationwide, critical open questions remain around how to most effectively achieve this goal. A primary question that lies at the heart of computer science education, especially early in learners' exposure to the field, is the choice of what programming tools to use and how best to design interfaces and activities that facilitate learners having meaningful and productive learning experiences early in their computer science careers. These choices are consequential along a number of dimensions including how they affect learners' perceptions of and attitudes towards the field, their roll in shaping the emerging understandings that the representations underpin, and their suitability for preparing learners for future computer science instruction. While there are languages and tools that excel along some of these dimensions, trade-offs exist between these competing aspects that must be managed. Does using a language designed to engage programming novices do a poor job of preparing them for transitioning to more powerful, enterprise languages? Does the representational infrastructure of a specific language promote programming practices that are incommensurate with those of other languages? Understanding the relationship between the languages and tools used in introductory programming experiences is critical, especially at this point in time, as the field is undergoing transition and rapid growth. There is potential that the tools, environments, and practices that are adopted now will become standardized and achieve

widespread use as computational thinking generally, and programming more specifically, increasingly become a part of the formal education landscape.

One approach to the design of introductory programming tools that is becoming increasingly widespread is the use of graphical, blocks-based programming environments that leverage a primitives-as-puzzle-pieces metaphor. In such environments, learners can assemble functioning programs using only a mouse by snapping together instructions and receive visual feedback on how and where commands can be used and if a given construction is valid. The use of this programming modality has become a popular feature of introductory computer science curricula and programming interventions targeted at young learners. Despite its growing popularity, open questions remain surrounding the effectiveness of blocks-based programming for helping students learn basic programming concepts and the overall effectiveness of the approach for preparing learners for future computer science learning opportunities that rely on text-based languages. The goal of this dissertation is to shed light on these two open questions and then use that knowledge to design a new tool for learning to program that draws on the affordances of the blocks-based approach while addressing drawbacks of the modality. We will evaluate the new tool alongside blocks-based tools and text-based environments to more completely understand the relationship between programming representations and the understandings and practices they promote. In doing so, we seek to provide evidence to better inform educators and administrators who are tasked with making consequential decisions around how learners are introduced to computer science and, more generally, to contribute to our larger understanding of the relationship between a given structuration and the understandings and practices it supports.

Goals and Outcomes

The three main goals of this dissertation are:

1. To better understand the relationship between representations of programming concepts and learners' emerging understandings, practices and attitudes towards programming in the early stages of learning to program.
2. To produce evidence-based recommendations on features to look for and features to avoid when choosing introductory programming environments and languages based on their effects on students learning as well as the suitability for preparing students for future programming and computer science learning opportunities.
3. Provide a proof of concept that it is possible to create a hybrid textual/graphical programming language that pairs the low-threshold aspects of graphical programming tools that research has identified with the high-ceiling, text-based programming approach used in higher education and professional contexts.

These three goals inform the following set of research questions that I will pursue in this dissertation:

1. For text-based, blocks-based, and hybrid blocks/text programming tools, what is the relationship between the programming modality used and learners' understandings of

programming concepts? What programming practices do learners develop when working in each of these three environments? How does the representational infrastructure used affect students' perceptions of programming with respect to utility, authenticity, enjoyment, and overall attitudes? And for each of these questions, how do the answers differ across blocks-based, text-based, and hybrid blocks/text environments?

2. How do understandings and practices developed while working in introductory programming tools support or hinder the transition to conventional text-based programming languages? How is this different between text-based, blocks-based and hybrid blocks/text introductory environments? How do learners' understanding of and attitudes towards programming change as learners shift from introductory tools to more widely used, professional programming languages?
3. Can we design hybrid introductory programming environments that blend features of blocks-based and text-based programming that effectively introduce novices to programming and computer science more broadly? How does such an environment perform relative to blocks-based and text-based programming tools with respect to conceptual understanding, development of productive programming practices, and attitudinal, motivational, and engagement outcomes for learners?

By answering these questions, the result of this dissertation will be:

- A clear understanding of the relationship between programming modality and resulting understandings of programming concepts, the practices that develop, and the attitudes toward and perceptions of programming they promote in learners.
- Documenting the challenges learners face in transitioning from introductory programming environments to the tools used in later computer science educational contexts, including what concepts and practices successfully transfer and which do not.
- Evidence-based recommendations on features to look for and features to avoid when choosing introductory programming environments, languages, and tools.
- A demonstration of a hybrid text/block-based programming environment and whether or not such approaches offer advantages over either text or blocks based tools.

Having laid out the motivation, research questions, and goals of this dissertation, this proposal continues with a review of relevant literature that informs this work. This includes work on studying novice programmers and the challenges they face, design work on the creation of low-threshold programming tools, and research looking at the relationship between representations and the understandings they support. Next, the formal study is described in detail. The study is a three condition, quasi-experimental design that will be conducted in high-school classrooms. Each condition, along with the design work that leads up to it and the analysis the follows are described along with the methods, settings, and materials that make up the project.

Literature Review

A large body of research has informed the study we are proposing for this dissertation including work on challenges novices have on learning programming concepts, designing languages and environments for novice programmers, and research on the relationship between representations and the understandings they engender. In place of a lengthy review on the full body of work done studying novices learning to program, we instead choose to narrow our review to the design portion of this literature, specifically reviewing efforts to design languages, environments, and tools to support learning to program (for a reviews of novices learning to program more broadly, see Robins, Rountree, & Rountree, 2003; Pears et al., 2007). We begin this section by first reviewing literature on the relationship between representations and learning as it underpins the discussion of the design of novice programming languages and environments and studies evaluating their strengths and weaknesses that follows.

Representations and Learning

"The tools we use have a profound (and devious!) influence on our thinking habits, and, therefore, on our thinking abilities." (Dijkstra, 1982)

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 (2000) 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" (diSessa, 2000, p. 116 empahsis 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" (diSessa, 2000, p. 115). The recognition that mental activity is mediated by tools and signs is one of the major contributions of the work of Vygotsky (1978, 1986) who argued that it is the external world, what he calls the intermental plane, that gives rise to internal cognitive functioning (Wertsch, 1991). Adopting this perspective informs why it is so crucial to understand the relationship between a growing family of programming representations and the understandings and practices they promote.

The role of representations on 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 (Boroditsky, 2001; Luria, 1982; Ong, 1982; Scribner & Cole, 1981; Vygotsky, 1986; Whorf, Carroll, & Chase, 1956). As we are less interested in thought and natural language broadly but instead logical thinking coupled with symbolic formalisms, we focus our review on scholarship within the mathematics domains as they are more closely related to our research program.

As we are interested in external representations (in our case programs and programming languages), it is important to review research looking at representations and the purposes they serve before returning to our interest in representations and the role they place in cognition and learning. Palmer (1978), in his early work forming a cognitive framework for understanding representations states: "a representation is, first and foremost, something that stands for

something else" (p. 262-3). His framework makes a division between the *represented world* and the *representing world* with a correspondence existing between the two and that using representations involves processing these two worlds to determine the relations held between the two. This is categorized as a "symbol-systems" perspective (Nemirovsky, 1994; Sherin, 2000). Nemirovsky (1994), drawing on the work of Bhaktin (1981) and his distinction between a sentence and an utterance, developed a framework that makes a distinction between the symbol-system perspective of Palmer and what he calls *symbol-use*, shifting focus from the rules of the representational system to an emphasis on their use and the meaning they carry within a particular in context. This emphasis on symbols-in-use and a recognition that learners' own knowledge and experience should influence the representations used and how they are studied and evaluated has been a recurring idea within the Learning Sciences (Confrey & Smith, 1994; diSessa, Hammer, Sherin, & Kolpakowski, 1991; Lave, 1988; Noss & Hoyles, 1996; Sherin, 2000; Weintrop & Wilensky, 2014; Wilensky, 1995).

As our interest is in student learning with representations, taking a perspective that moves past the symbol-system in isolation is essential as "a symbol systems analysis does not, in its raw form, provide a theory of the knowledge or capabilities possessed by students. Instead, it describes knowledge only by the function that it must perform" (Sherin, 2000, pp. 405–6). Sherin, having identified this gap between the symbolic systems view and the understanding they promote and usages they enable, pursued a research course to better understand this relationship that is similar to our own. Focusing on concepts from physics and investigating the use of conventional algebraic representations as compared to programmatic representations, Sherin (2001a) found that different 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" (Sherin, 2001a, p. 54).

Wilensky and Papert (2006, 2010) give the name structuration to describe this relationship between the representational infrastructure used within the domain and the understanding that infrastructure enables and promotes. While often assumed to be static, Wilensky and Papert show that the structurations that underpin a discipline can, and sometime should, change as new technologies and ideas emerge. In their formulation of the idea of structurations, Wilensky and Papert (ibid) document a number of restructurations, shifts from one representational infrastructure to another, and provide a set of criteria with which to evaluate them. Shifts including the move from Roman numerals to Hindu-Arabic numerals (Swetz, 1989), the use of the Logo programming language to serve as the representational system to explore geometry (Abelson & DiSessa, 1986), and the use of agent-based modeling to representation various biological, physical, and social systems (Blikstein & Wilensky, 2009; Wilensky, 1999; 2001; Wilensky & Reisman, 2006) along with the aforementioned studies of Sherin's, all serve as examples of this type of representational shift. This work highlights the importance of studying representational systems, as restructurations can profoundly change the expressiveness,

learnability, and communicability of ideas within a domain. As we will see in the next sections, the rise of new programming modalities, representations, and tools demand that such analyses be conducted to better understand the effects of these emerging approaches to teaching, learning, and using ideas within the domain of computer science.

Novice Programming Environments

The previous section highlighted the critical importance of representations and their influence on cognition and learning, with that as a backdrop, we now proceed with a review of various design efforts indented to improve a learner's introduction to the field of computer science and the practice of programming. A great deal of work has been done on the design and implementation of programming languages and environments for beginners (for reviews of this work, see: Duncan, Bell, & Tanimoto, 2014; Guzdial, 2004; Kelleher & Pausch, 2005). In this section we discuss some of the more influential languages and environments and theoretical contributions that informed the environments and designs being investigated in this dissertation.

Languages and Environments from constructionist tradition

Constructionism has a long history of producing computer-based learning environments that empower learners and laid much of the important theoretical groundwork upon which current movements to promote programming and computer science more broadly are built. Languages and environments from the constructionist tradition have successfully enabled children (as well as adults) to construct their own, personally meaningful computational artifacts, often with little (or no) formal instruction. In this section we provide a brief history of the more influential constructionist programming environments, beginning with Logo, the language that started it all.

Logo

The Logo programming language was iteratively developed by Seymour Papert and colleagues in Boston in the 1960's. Logo was the earliest programming language designed explicitly for children (an early report is given in Feurzeig, Papert, Bloom, Grant, & Solomon, 1969). Based on the Lisp, "Logo was designed to provide a conceptual foundation for teaching mathematical and logical ways of thinking in terms of programming ideas and activities" (Feurzeig, Papert, & Lawler, 2011, p. 487). In Mindstorms, Papert (1980) dedicates a chapter to discussing the theoretical roots that most directly informed the design and creation Logo. The first stems from Piaget and his work as epistemologist, recognizing that to study how a child comes to understand a concept is to study the concepts itself. Logo, in its design to teach mathematics in a fundamentally different way, reimagines what mathematics looks like and how the learner interacts with and thinks with mathematical concepts. The second theoretical influence to Logo was from the field of artificial intelligence (AI). As one of the goals of AI is to build machines that can perform intelligent behavior, such an endeavor needs to study the nature of intelligence. An appeal of this work was that its methodology relies heavily on computation and computational environments that force theoreticians to concretize and explicitly represent their ideas and theories of learning by computationally reifying them. Papert saw in this line of

work the potential for giving children the opportunity to similarly think concretely about mental processes and what it means to learn.

Logo was designed with the principle of "low threshold, no ceiling" and saw early and widespread international adoption and influence in the mathematics community especially among forward thinking educators. A central contribution of Logo to introductory programming design was the invention of the turtle – an entity (either physical or virtual) that the user defines instructions for in the form of movement instructions, then watches the turtle carry them out. The turtle leveraged what Papert (1980) called "body sytonicity" which enabled learners to use their own experiences in the world as a productive resource for generating programming instructions. As we will see throughout this review, this design feature shows up repeatedly as an accessible way for learners to engage with and have early successes with programming.

Naïve Realism and Spatial Metaphors with Boxer

Boxer (diSessa & Abelson, 1986) was an early descendant of Logo that added to the environment the feature that every object in the system had an on-screen graphical representation that could be inspected, modified and extended. This design feature was based on the naïve realism theory of mind and was intended to create a programming interface where "users should be able to pretend that what they see on the screen is their computational world in its entirety" (diSessa & Abelson, 1986, p. 861). As such, the environment presents the user with a complete visual model of what is happening in the machine. This resulted in a design where all computational objects in Boxer are displayed as boxes (hence the name). These boxes can each be unpacked, giving the users access to its contents, creating a "glass-box" environment where nothing is hidden from the learner. A second major design feature of Boxer was the use of a spatial metaphor as a way to display information about the entities within a program and their relation to each other. As such, boxes visually rendered inside other boxes spatially depict a hierarchy of boxes. This use of visual layout of commands as a means of communicating information about the program and the effect of such an interface will resurface again in later environments and is at the heart of the questions being pursued in this study.

Many-Turtled Logo Environments

A second direction that the Logo language was taken was to relax the constraint that there can be only a single entity being controlled in the environment. By allowing users to introduce as many turtles as they want and providing tools that allow them to give instructions to only a subset of the turtles, learners could create programs that support the investigation of emergent and decentralized complex systems (Wilensky & Resnick, 1999). Two early version of these environments were StarLogo (Resnick & Wilensky, 1993) and StartLogoT (Wilensky, 1997). The successor to StartLogoT, NetLogo (Wilensky, 1999b) has since become a very widely used implementation of a Logo-based multi-agent programming environment.

Building on the finding that programming and construction are effective ways for learners to develop mathematical understandings, these environments have extended this work beyond mathematics to include a wide range of STEM topics. NetLogo was designed as a modeling environment that captures emergent phenomena (Wilensky, 2001). NetLogo enables

learners to use, modify, and create models of real-world phenomena as a means to develop deep understandings of the underlying mechanisms and properties of the topic under investigation. A core constructionist design principle of NetLogo is that by programming models of scientific phenomena, students will learn science more deeply while also learning programming and modeling. This approach has been found to be effective for teaching students in a wide variety of domains including biology (Wilensky & Reisman, 2006), electromagnetism (Sengupta & Wilensky, 2009), chemistry (Levy & Wilensky, 2011; Stieff & Wilensky, 2003), evolution (Wilensky & Novak, 2010) and material sciences (Paulo Blikstein & Wilensky, 2009). By situating the programming activity within a larger goal of learner specific content introduces another oft-replicated feature of introductory programming environment – the importance of context surrounding the programming activity (Cooper & Cunningham, 2010; Guzdial, 2010).

Blocks-based and Graphical Logo Environments

The last Logo descendants we include in this review are environments that use a blocks-based or other graphical programming approach. Scratch (Resnick et al., 2009) is the most well known of the group, but other blocks-based constructionist tools include LogoBlocks (Begel, 1996), StarLogo TNG (Begel & Klopfer, 2007), and Snap! (Harvey & Mönig, 2010). These environments replace the textual interface of Logo with a programming-primitive-as-puzzle-piece metaphor that allows users to drag commands into place and snap them together to assemble their programs. Other approaches that leverage similar graphical approaches include ToonTalk (Kahn, 1999), which relies on a much more direct visual metaphor for defining instructions, and Squeak (now e-toys) (Kay, 2005), which uses a rules-as-tiles programming mechanism. We only briefly mention these environments here, as we dedicate more time to graphical and blocks-based programming tools later in this literature review.

Languages and Environments from outside the Constructionist community

While many early programming languages emerged from the constructionist research community, the computer science education community also developed a number of programming languages and environments designed for novices. Here we review some of the more influential efforts that helped inform this dissertation. This includes languages designed explicitly for educational contexts and software authoring tools for novices.

Beginner Programming Languages

Early on it was recognized that the design of the language itself can support or hinder students in their quest to master programming, which resulted in early efforts to develop more accessible programming languages (Mendelsohn, Green, & Brna, 1990). Along with Logo, which was explicitly designed with mathematics learning in mind, other languages emerged with the goal being to serve as introduction to the field of computer science. An early, influential language designed for novices was BASIC (Kemeny & Kurtz, 1980), 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. BASIC experienced a great deal of success and was a popular language throughout the early ere of personal computing from the mid 1970's through the 1980s.

As the field of computer science education matured, new languages and strategies emerged that were designed to serve as introductory tools and prepare learned for more industrial, fully featured languages. Languages such as Blue (Kölling & Rosenberg, 1996) and JJ (Motil & Epstein, 1998) simplified syntax and provided tools to allows learns to focus on programming fundamentals before progressing to fully featured languages. Other languages tried to blend the best features of various languages in hopes of developing new languages that were both powerful and easy to learn and use (Holt & Cordy, 1988). Another direction introductory programming languages took was to create more declarative languages in which programming was a more direct, rule defining activity. Languages such as Prolog (Colmerauer, 1985), and later graphical environments such as Agentsheets (Repenning, Ioannidou, & Zola, 2000), ToonTalk (Kahn, 1999) and StageCast Creator (Smith, Cypher, & Tesler, 2000) utilize this strategy.

A third approach was the use of mini-languages, which are small, simple languages designed to support the first steps in learning to program (Brusilovsky, Calabrese, Hvorecky, Kouchnirenko, & Miller, 1997). Mini-languages often centered 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 (Pattis, 1981). These mini-languages share features with domain-specific languages, which are not intended for general purpose programming, but instead tailor a smaller language around specific tasks, narrowing the gap between the objective and the representations in which intentions are encoded (Van Deursen, Klint, & Visser, 2000).

Another approach taken in the design of programming languages for novices is to bring the programming language closer to natural language. The first language that tried to draw on natural language grew out of an effort to create a "Common Business Language" (COBOL), which intentionally tried to maximize the use of English in its syntax (Sammet, 1981). Another early language that took this approach was Hypercard. When asked about Hypercard's ancestors, designer Bill Atkinson responded: "The first one is English. I really tried to make it English-like" (*Goodman*, 1988 as cited in Bruckman & Edwards, 1999, p. 208). A more recent language that adopted this strategy, that was designed explicitly for younger learners is the MOOSE programming language designed to enable kids to create places, creatures and other objects in a text-based virtual game (Bruckman, 1997).

A final strategy that is important to include in this review of approaches to designing programming languages for novices 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, is a serious barrier for novice programmers (Denny, Luxton-Reilly, Tempero, & Hendrickx, 2011; Stefik & Siebert, 2013). Through a serious of controlled experiments that had novices use one of a variety of languages that demonstrated various syntactic features, Stefik and Siebert (2013) found that characteristics of syntax do

directly influence a languages learnability. One solution to the syntactic problem that the creation of programming tools that prevent syntax errors which can be accomplished through the use of visual cues and graphical composition tools. This features is especially relevant to the proposed study, as graphical programming proponents boast that the lack of syntax is a key features that contributes to its appropriateness for young learners (Resnick et al., 2009), but research is finding this approach does not solve the syntax problem, but only delays it (Parsons & Haden, 2007; Powers et al., 2007). This issue will be discussed in more detail later in our literature review.

Integrated Development Environments for Novices

Along with recognizing that features of the language can support or hinder learnability, it was realized that software used to author programs (called Integrated Development Environments or IDEs) themselves could provide a large number of supports the help the novice overcome challenges including syntax errors, deciphering compilation errors, and problematic sections of programs. This recognition coincided with a larger shift in the computing space that was advocating from a shift away from users conforming to computer requirements towards a world were the computer conformed to the user (Norman, 1993). These efforts initially focused on supporting experts, but educational technology designers soon realized that what is good for the expert is often not the same as what is best for the novice and when designing educational tools, you should proceed with the learner in mind (Soloway, Guzdial, & Hay, 1994). As such a growing number of IDEs have been developed explicitly with an eye towards supporting novice programmers.

An early and influential development environment designed to facility novices specifically through a reduction on potential syntax errors was the Cornell Program Synthesizer, which built on the fact that programs are not flat text, but instead "hierarchical compositions of computational structures and should be edited, executed, and debugged in an environment that consistently acknowledges and reinforces this viewpoint" (Teitelbaum & Reps, 1981, p. 563). This tool provided the users with templates that followed the syntactic structure of the language and thus, when filled in, would result in valid statements that could be added to the program. These templates were constructed by following the grammar defined by the language's abstract syntax tree (AST). While a number of different project and research groups followed the lead set by the Cornell Program Synthesizer, Carnegie Mellon University emerged as a leader in the development of programming tools that used features of the language to support novices. Over the course of a number of projects, the CMU group iteratively developed a family of programming environments including GNOME (Garlan & Miller, 1984), Genie (Chandhok & Miller, 1989), and ACSE (Pane & Miller, 1993). These environments progressively introduced features including code layout based on the language's AST, incremental parsing and feedback for syntax errors while editing, supporting multiple views of the same program simultaneously including high-level design views and code navigation views, and runtime visualization tools (a history of these environments can be found in Miller, Pane, Meter, & Vorthmann, 1994). Differentiating these tools from the efforts in the previous section is the fact that being

environments, these tools were not necessarily tied to a specific language. For example, GNOME environments were created for various languages including Karel the Robot, Pascal, Fortran, and Lisp (Miller et al., 1994). The inclusion of a language's AST as part of what determines how programs are composed is central to the types of visual programming tools of interest to this study and an idea we will return to later in this literature review.

More recently, a new generation of IDEs have been developed that are designed with a specific language in mind and include features unique to that language and even to specific pedagogies for learning that language. Environments such as DrScheme developed for the scheme programming language (Findler et al., 2002), and DrJava, a similar tool developed for Java (Allen, Cartwright, & Stoler, 2002), present an integrated graphics-rich editor and use a functional read-evaluate-print development cycle to assist novices in their early programming endeavors. The BlueJ environment is a popular Java IDE designed to support an object-first approach to learning to program in Java (Kölling, Quig, Patterson, & Rosenberg, 2003). BlueJ was intentionally designed to keep the interface simple as to not overwhelm the learner and emphasize the features of the language deemed most important, which in BlueJ is the object-oriented nature of Java. Building off of the successes of BlueJ and remaining faithful to the learned-focused design, the BlueJ team release a second IDE called Greenfoot, designed for younger learners that adds visual program execution to the IDE to further support younger learners and their developing understanding of programming concepts (Henriksen & Kölling, 2004). A particular feature of Greenfoot that is of relevance to this study is the decision to share many interface features between BlueJ and Greenfoot to make transition between the environments easier for learners as they progress. As we discuss below, transition from introductory to more sophisticated programming environments and language is rarely a consideration for designers of IDEs for novices.

Visual Programming

As the development of programming languages and environments evolved, it was found that shifting from an all-text representation of programs to an approach the leverage spatial and graphical features could be productive for learning and understanding (Smith, 1977). Collected under the label "Visual Programming", these environments are broadly defined as "any system that allows the user to specify a program in a two (or more) dimensional fashion" (Myers, 1990, p. 98). While this definition is intentionally general, it excludes text-based programming (which is considered to allow composition of programs in a single, horizontal dimension), software used to produce visualizations (like animation and drawing programs), and tools that visually depict the execution of programs (like environments that visually render memory contents or algorithmic flow of a running program). To provide a framework for evaluating visual programming environments, Green and Petre (1996) developed a cognitive dimensions framework that characterizes features of these environments and maps out the trade-offs that exist between different visual design choices. These cognitive dimensions include entries such as Abstraction Gradient (various granularities of abstraction supported), Consistency (how formulaic is the language), Progressive Evaluation (what feedback is available from partially-

complete programs), and visibility (how easy is it see and read the code). This framework proved to be effective and is widely used to evaluate visual programming environments.

Direct manipulation was an early and widely implemented approach to graphical programming that touches on a number of strengths of a graphical authoring modality. Hutchins et al. (1985) explain the concept:

> The promise of direct manipulation is that instead of an abstract computational medium, all the "programming" is done graphically, in a form that matches the way one thinks about the problem. The desired operations are performed simply by moving the appropriate icons onto the screen and connecting them together. Connecting the icons is the equivalent of writing a program or calling on a set of statistical subroutines, but with the advantage of being able to directly manipulate and interact with the data and the connections. There are no hidden operations, no syntax or command names to learn. What you see is what you get. (p. 314)

Included in this definition are a number of key features of graphical programming: the presence of onscreen icons that carry some computational or programmatic meaning that can be controlled directly, a two (or more) dimensional space to work within, a minimization (or absence) of syntax or commands, and a general transparency that permeates the environment and how it is meant to be used. Based on the promise of easier to learn, easier to use programming systems, many visual programming languages have been designed and evaluated. Direct manipulation tools often rely on flow-chart or data-flow diagrams that map logical flows through instructions (Hils, 1992). LabVIEW (Johnson, 1997; Santori, 1990), a circuit diagram program built on an electronic block wiring metaphor, often servers as an exemplar direct manipulation environment in studies of this programming modality, with the results of these studies generally being mixed (Whitley, 1997). Flogo (Hancock, 2003), a graphical programming environment designed to facilitate learners programming robot behaviors, used a visual dataflow view of information intended to make programs more understandable, accessible, and modifiable. A more contemporary version of direct manipulation software is the Lego Mindstorms NXT-G programming tool (Lego Systems Inc, 2008), which allows children to program robots by dragging iconic representations of program commands and robot components from a palette onto a workspace, where they can be wired together.

Another family of programming tools that emerged from this tradition use images and a graphical stage rendered as a grid to allow users to program rules the system can follow. Building on the idea of programming-by-demonstration, KidSim, later renamed Stagecast Creator (Smith, Cypher, & Spohrer, 1994; Smith et al., 2000), was developed to allow users to create games by defining rules using symbols and graphics, removing the need for syntax or text-based programming commands. These graphical rules govern the behavior of the entities (called agents) in the world being programmed making it easy to create playable video games. Repenning and colleague took a similar approach (although they choose to call it programming-by-problem-solving) in the development of Agentsheets, an environment in which users program sets of agents that move via graphical rules (Repenning & Sumner, 1995). Unlike Stagecast

Creator, Agentsheets moves beyond game-making to include the creation and exploration of scientific models and simulations as part of its uses (Repenning et al., 2000). With the release of Agentcubes, the two dimensional restriction of the stage has been lifted, enabling learners to create three-dimensional games and simulations (Ioannidou, Repenning, & Webb, 2009).

Evaluating Visual Programming Environments

In the early 1990s a thread of researchers, lead by Green and Petre among others, systematically compared text-based and visual programming to understand which was superior, While some studies showed promise in the use of visual programming tools (Baroth & Hartsough, 1995), other studies conducted in more controlled environments found contradictory evidence. Green and collaborators found that visual programming environments required longer amounts of time to develop solutions and provided less guidance on strategic approaches resulting in more difficulty among novice programmers (Green & Petre, 1992; Green, Petre, & Bellamy, 1991; Petre, 1995). They attributed these findings to unfamiliarity with available secondary notations of the languages (a dimension from the Green and Petre's cognitive dimensions framework that captures the use of layouts and other informal cues to express structure) and the match-mismatch hypothesis (Green, 1977), which links difficulty in generating function solutions to misalignment between the structure of a problem with the structures supported by the language. These findings were reproduced in a later set of studies with a larger set of visual programming tools, in which it was found that various visual representations were at best on-par with their textual equivalents (Moher et al., 1993). For a longer review of this work, see Blackwell et al. (2001). While much of this comparative work was conducted over twenty years ago, the field is still active with studies being conducted with new tools (for example Hundhausen, Farley, & Brown, 2009). We will return to these more contemporary studies later as they focus not just on comparisons between tools, but also transitioning between representations.

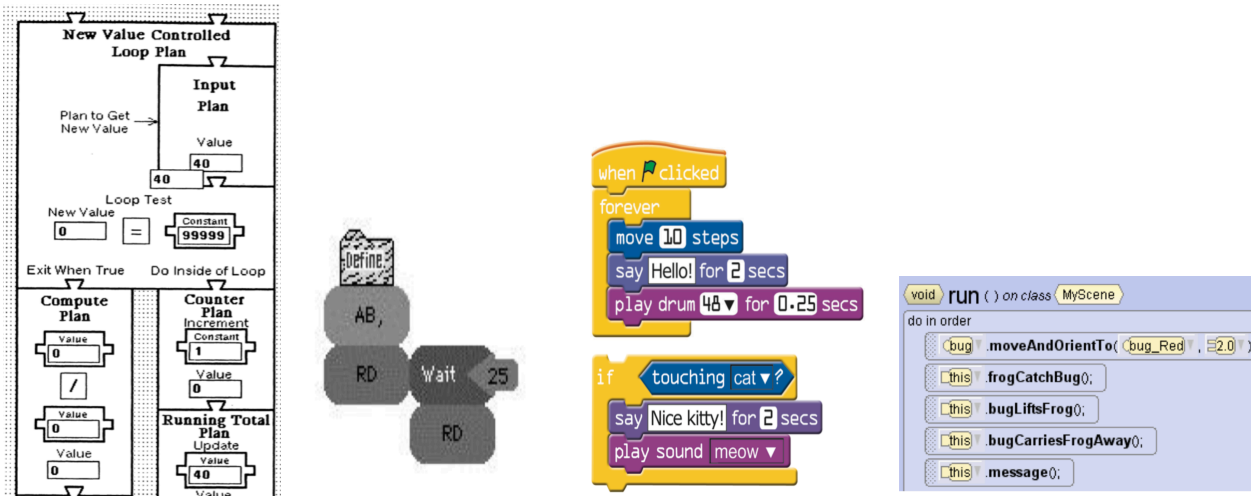
Two other approaches to visual programming are important to mention. The first is the use of tangibles as representations of programming statements, an approach we briefly discuss in our section on efforts to diversify the field of computer science. The second consist of languages that do not rely on visual metaphors of rules or objects, but visual metaphors of programming statements and abstractions directly. We call these blocks-based programming environments and review them in more detail in the section that follows.

Blocks-based Programming

The blocks-based approach of visual programming, 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 (Resnick et al., 2009), Snap! (Harvey & Mönig, 2010), and Blockly (Fraser, 2013). These programming tools are a subset of the larger group of editors called *structured editors* (Donzeau-Gouge, Huet, Lang, & Kahn, 1984) that make the atomic unit of the composition tool a node in the AST, as opposed to a smaller piece (i.e. a character) or a larger piece (a fully formed functional unit). In making these AST elements the building blocks,

then providing constraints to insure nodes can only be added to the programs AST in valid ways, the environment can prevent against syntax errors. The constraints can be provided in a number of ways. 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 scripting 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 preventing syntax errors but retaining the practice of assembling programs instruction-by-instruction. Along with using block shape to denote usage, there are other visual cues to help programmers, including color coding by conceptual use, and nesting of blocks to denote scope (Tempel, 2013).

Early version of this interlocking approach include LogoBlocks (Begel, 1996) and BridgeTalk (Bonar & Liffick, 1987) which helped formulate the programming approach which has since grown to be used in dozens of applications. Alice (Cooper, Dann, & Pausch, 2000), an influential and widely used environment used in introductory programming classes, uses a very similar interface and has been the focus of much scholarship evaluating the merits of the approach. Figure 1 shows programs written in a number of blocks based programming tools.



(a) (b) (c) (d)

Figure 1. Four example blocks-based programming languages: (a) BridgeTalk, (b) LogoBlocks, (c) Scratch, and (d) Alice.

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 programing across a variety of domains including mobile app development with MIT App Inventor (Wolber, Abelson, Spertus, & Looney, 2011) and Pocket Code (Slany, 2014), modeling and simulation tools including StarLogo TNG (Begel & Klopfer, 2007), DeltaTick (Wilkerson-Jerde & Wilensky, 2010), and EvoBuild (Wagh & Wilensky, 2012), creative and artistic tools like Turtle Art (Bontá, Papert, & Silverman, 2010), and PicoBlocks (*PicoBlocks*, 2008), commercial educational programming applications like Tynker (*Tynker*, 2014) and Hopscotch (*Hopscotch*, 2014), and game-based learning environments like RoboBuilder

(Weintrop & Wilensky, 2012), Lightbot (Yaroslavski, 2014) and the activities included in Code.org's Hour of Code (*Hour of Code*, 2013) and Google's Made with Code initiative (*Made with Code*, 2014). Further, a growing number of libraries are being developed that make it easy to develop application or task specific blocks-based languages (Fraser, 2013; Roque, 2007). This diverse set of tools and the ways the modality is being used highlights its recent popularity and speaks to the need for more critical research around the affordances and drawbacks of the approach.

Evaluating Blocks-based Programming Environments

In our review of literature focusing on the educational efficacy of blocks-based languages, we focus on Scratch and Alice, as these two tools have the widest use in contemporary computer science education of the blocks-based environments listed above. 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 different target age groups. Scratch from its inception, was focused on younger learners and informal environments (Resnick et al., 2009), while Alice was targeted at more conventional computer science educational contexts and, as such, has a been the focus of more initiatives to evaluate student learning of programming concepts (Cooper et al., 2000).

We begin by reviewing literature on Scratch investigating its use as the language of choice in formal computer science environments. Ben-Ari and colleagues have conducted a number of studies of the use of Scratch for teaching computer science. Using activities of their own design (Armoni & Ben-Ari, 2010), Meerbaum-Salant et al. (2010) concluded 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 from this project, a closer look at the programming practices learners developed while working in Scratch gave pause to the excitement around the results. These researchers found that students developed unfavorable habits of programming, including a totally bottom-up programming approach, a tendency for extremely fine-grained programming, and often incorrect usages of programming structures as a result of learning programming in the Scratch environment (Meerbaum-Salant, Armoni, & Ben-Ari, 2011). Other work looking at comparing blocks-based to text-based programming using Scratch has similarly 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 (C. M. Lewis, 2010). Given Scratch's intention on being used in informal spaces and it's emphasis on introducing diverse learners to programming, it is important to highlight Scratch's success in generating excitement and engagement with programming among novice programmers (Malan & Leitner, 2007; Maloney et al., 2008; Tangney et al., 2010; Wilson & Moffat, 2010).

Compared to Scratch, the Alice programming environment has a longer history of serving as the focal programming tool in introductory programming courses. Much of the motivation for using Alice in courses is based on findings that Alice is more inviting and engaging than text-based alternatives, and improves student retention in CS departments (Johnsgard & McDonald,

2008; Moskal, Lurie, & Cooper, 2004; Mullins, Whitfield, & Conlon, 2009). Alice has also effectively been used by instructors who adopt an object-first approach to programming as it provides an intuitive and accessible way to engage with objects with little additional programming knowledge needed (Cooper, Dann, & Pausch, 2003). Part of Alice's success and relatively widespread use is due to the fact that the authors of Alice have authored a number of textbooks and curricula that can serve as texts for an introductory programming course (Dann, Cooper, & Ericson, 2009; Dann, Cooper, & Pausch, 2011). It is also important to mention here the growing body of work looking at blocks-based programming as an introductory tool in service of preparing students for learning more conventional text-based programming, which we discuss in the following section. Until recently more research had been conducted around the transition from Alice to Java, as it is more frequently featured in conventional CS contexts, but recently this line of research has expended to include Scratch and other blocks-based tools. As this goal is at the heart of this dissertation, we devote a full section to reviewing efforts towards this end.

From Blocks-based to Text-based Programming

With the rise in popularity of the blocks-based approach to programming, a question of growing importance is how well these tools prepare students for future, text-based programming languages. Do students develop understandings that can serve as a foundation for future learning or do students struggle to apply what they have learned in new programming contexts with more powerful, text-based programming languages. Recent studies have begun to explore this question of transfer of programming knowledge between blocks-based and text-based programming. Before reviewing this literature, it is important to note that not everyone is in agreement that blocks-based programming is indeed only to be used a stepping-stone for text-based programming. Some argue that a blocks-based modality is a sufficient end-point for those who are not intent on pursing a career in programming (Modrow, Mönig, & Strecker, 2011). While we see merit to this position, as we are focused on high-school aged students, and all widely adopted computer science assessments (notable the AP Computer Science exam) are conducted with text-based programming languages and a vast major of libraries and programming tools use a text-based approach, we see this question of great importance. We begin this review by looking at studies attempting to bridge Alice and Java as it has the longest, and most well documented history.

From the outset, Alice was intended for formal educational contexts and was widely shared and discussed in computer education circles. As a result, it was has become a popular tool for use in introductory computing courses at the university level, thus the challenge of transition from Alice to Java has become as an active area of research. As is often the case, results have been mixed in studies looking at the transition of students from Alice to Java. Powers et al. (2007), in their study following students from a semester in Alice to a semester in Java (using BlueJ) documented a number of challenges faced by their students including issues with syntax, frustration with lack of progress at a similar pace as in Alice, and a feeling that programming in Alice was not authentic due to its graphical nature. This position has been taken to various

degrees (Cliburn, 2008), with some researching going so far as to state "based on our classroom experience, we question its real pedagogical value for programming education at the tertiary level. Students do not seem to naturally make a strong connection between the formal coding process and what they are doing with Alice" (Parsons & Haden, 2007, p. 213). Another study compared students spending time using Alice compared to students working through the same activities in pseudo-code and found that students in the pseudo-code condition performed better on standard performance measures (Garlick & Cankaya, 2010). In contrast, other researchers have found Alice to be an effective way to introduce learners to programming and had success using it as a tool for transition to Java. Notably, the authors of Alice developed a tool that allowed students to move back and forth between Java and Alice which was found to be effective at bridging this gap (Dann et al., 2012). Citing this work, classes have adopted this strategy of mixing Java and Alice as a way to leverage the strengths of the visual programming approach while mitigating issues cited above (Dann et al., 2009). A number of textbooks have also been written to bridge the gap (Adams, 2007; Dann et al., 2009; J. Lewis & DePasquale, 2008), but research evaluating the effectiveness of these texts is relatively sparse.

Beyond Alice, a growing number of tools are being designed to address the blocks-to-text gap, either as new stand-alone tools or add-ons to existing tools. Such efforts including the ScratchBlocks (*ScratchBlocks*, 2014) and SLASH (Behnke, 2013) add-ons to Scratch, the TAIL plugin (Chadha, 2014) and the App Inventor Java Bridge (*App Inventory Java Bridge*, 2014) for MIT App Inventor. PicoBlocks (*PicoBlocks*, 2008), TurtleArt (Bontá et al., 2010), and recently the Tynker platform (*Tynker*, 2014) all come with the ability to view text-based equivalents to programs constructed with the blocks-based interface. Other newer tools provide native language translation, for example, the Blockly toolkit comes with built-in language generators that allow you to convert graphical scripts to equivalent JavaScript, Python, or XML files (Fraser, 2013). Additionally, Blockly is architected in such a way as to make it easy to add additional generators to the library making it extensible for future blocks to text transformations. The DrawBridge project is noteworthy in it's effort to bridge blocks-based and text-based programming by introducing pen-and-paper drawing and program-by-demonstration features into its larger pedagogical strategy (Stead & Blackwell, 2014). Game authoring has also been used as a context to motivate blocks-to-text programming as demonstrated by the Flip project, although this environment's text-programming uses natural language expressions over conventional text-based programming syntax (Howland & Good, 2014). These projects are all recent efforts and as such, literature on the general effectiveness of various tools is only starting to be published. It is also important to note the difficulty in transferring between graphical and text-based programming environments should not be surprising as researchers have documented difficulties in novices transferring knowledge between two text-based programming languages (Scholtz & Wiedenbeck, 1990; Wiedenbeck, 1993), so seeing similar difficulties across modalities is unsurprising.

Diversifying Computer Science through Novel Environments and Activities

The field of computer science has historically struggled with both gender and racial diversity. The most recent numbers from the annual Taulbee Survey that tracks enrollment in computer science related disciplines find that only 14.5% of bachelor's degrees awarded in 2013 went to women, while 6.5% went to Hispanic students and only 4.5% were to Black or African American students (Zweben & Bizot, 2014). The male-dominated nature of the computing field and the culture that has emerged are well documented and many interventions have been proposed to try to address it (American Association of University Women, 1994; Fisher, Margolis, & Miller, 1997; Margolis & Fisher, 2003). Similarly that lack of racial diversity in the field has also been the focus of much scholarship (Margolis, 2008). To address these issues of underrepresentation, the literature recommends the creation of tools and activities specifically designed to attract and engage learners from these underrepresented populations by drawing on areas of interest and cultural relevance (Bruckman, Jensen, & DeBonte, 2002; DiSalvo, Guzdial, Bruckman, & McKlin, 2014).

One such approach blends hands-on crafts and computing to engage learners. The Lilypad Arduino (Buechley & Eisenberg, 2008) is a fabric based construction kit that enables novices to design and build their own e-textiles. In doing so they encounter fundamental computing and engineering ideas in a more creative, inviting and enjoyable context that results in learners reporting high levels of engagement with interest in computing (Buechley et al., 2008). The use of hands-on, tangible interfaces has also been found to be one successful approach for engaging female learners in other programming activities that do not share the Lilypad Arduino's creative component (Horn et al., 2009).

A second approach to attracting and engaging underrepresented populations in computing builds on the practice of storytelling (Burke & Kafai, 2010; Papadimitriou, 2003). One successful tool that builds of storytelling is Storytelling Alice. As the name suggests, Storytelling Alice is a version of Alice designed to support storytelling as its central activity. Studies comparing Storytelling Alice to conventional Alice (which lacks some storytelling support features) found that girls using Storytelling Alice were more motivated to program and spent longer working on their programming projects (Kelleher, Pausch, & Kiesler, 2007). Finally, as previously discussed, studies of Scratch in informal afterschool programs has been found to be an effective way to engage urban youth as the tool is flexible enough to allow learners to integrate aspects of their own lives into the programs they develop (Maloney et al., 2008).

Software Design: Three Introductory Programming Environments

Having reviewed the literature on introductory programming tools, identified challenges the field currently faces, and reviewed various strategies and attempts to address these challenges, we now continue by presenting the three programming environments that sit at the heart of the proposed study. This dissertation takes the form of a three-condition quasi-experimental study in which three different programming environments are used by students working through the same set of activities as a way to study the effects the representational tools have on learners. As such, the first major effort of this study is the design and implementation of these introductory programming environments that can serve to evaluate the role of

programming modalities on the learner. In this section we introduce the three environments we will develop before presenting the formal study design.

Briefly, the three environments will be browser-based programming tools that use either a text-based, a blocks-based, or a hybrid blocks/text representation for learners to interact with. The goal in the design of these environments is threefold: 1) to provide authentic interactions with programming concepts using the programming modality of each environment 2) to support activities and interactions consistent with the most recent generation of introductory programming environments, and 3) as much as possible, hold features of the environments constant, including the capabilities of the tools, the look and feel of the user interface, and characteristics of the runtime environment. The intention is that the only difference between the three environments will be how programs are visually represented (graphical or textual) and the actions the learners take in composing programs (typing versus dragging).

The three environments are all derived from the snap! programming environment (Harvey & Mönig, 2010). Snap! is a JavaScript-based implementation of a blocks-based programming language that can run in any modern web browser. Programs in snap!, like other graphical blocks-based programming environments (such as Scratch and Alice), center around controlling on-screen avatars, called sprites, as they move around a stage. Programming in these environments largely entail giving behavior to sets of interacting sprites to create animations, games, and stories. The snap! user interface (Figure 2) is broken down into five main sections: the Palette where language commands are organized, the Scripting Area where programs are composed, the Stage where programs are visually carried out, the Sprite Corral where the available set of sprites are displayed, and the Tool Bar which provides menus for additional capabilities (like saving and loading programs).

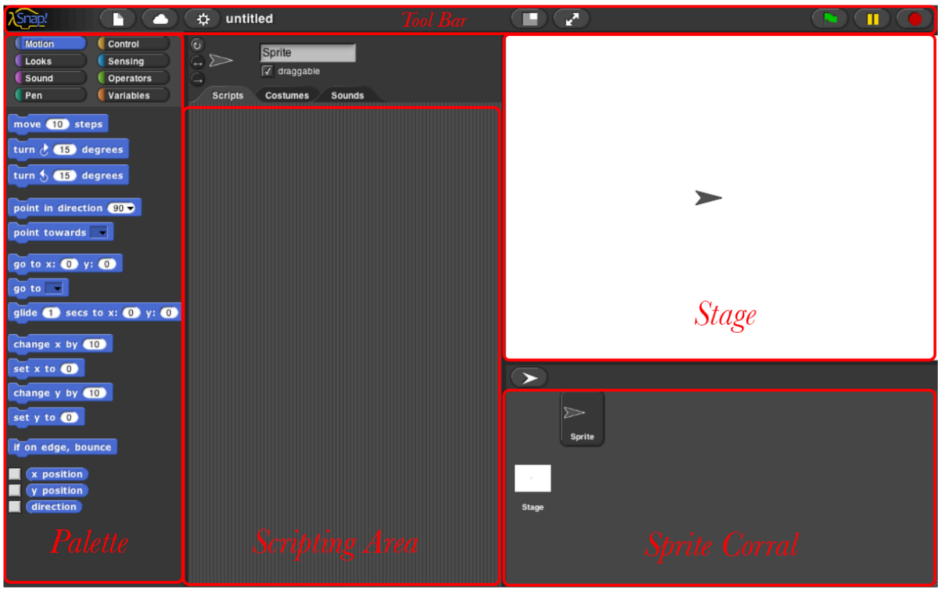


Figure 2. The snap! interface with sections labeled.

As all three environments will be built from the same core library, thus they will use the same visual execution environment and, as much as possible, share language semantics, behavioral properties, and design aesthetics. This means each environment will have a Stage where programs are visually executed and center around defining behaviors for sprites. The three programming environments will be isomorphic, meaning anything that is possible in one language will be possible in the other two, and, will be able to be accomplished using roughly the same number of commands. All three environments will provide learners with the same set of programming primitives based on those provided by snap!. Table 1 provides information about the language.

Table 1. The language categories of snap! and information about each.

Category	Types of Blocks	Example Blocks
Motion	Blocks that control the movement of the on-screen sprites	`forward __ steps; turn __ degrees; glide to __`
Looks	Blocks that control the visual appearance of the sprite, including costumes, thought and speak bubbles, and visual effects	`next costume; say __ for __ seconds; change size by ___`
Sounds	Blocks that provide the ability to incorporate audio in a project	`play sound __; play note __ for __ beats`
Pen	A set of blocks that allow users to programmatically draw on the stage by leaving a trail behind the sprites as they move.	`pen down; set pen color to__; stamp; clear`
Sensing	Blocks that allows sprites to interact with other entities including the mouse, the user (via prompting for information), and other sprites	`touching __?; ask __ and wait; mouse down?`
Events	Blocks that can control when a given script executes	`When __ key is pressed?; When I receive __?`
Control	Blocks concerned with flow of control as well as cloning sprites	`repeat __; while__; if__; report__`
Operators	A set of blocks providing logical operators, Boolean primitives, mathematical functions, and string manipulations	`__ + __; __ < __; __ and __; true; length of __`
Variables	Blocks for creating and using variables	`make variable; change __ by __`
Lists	Commands for creating, editing, and using collections of elements	`item __ of __; insert __ at __ of __;`

The Blocks-based Environment

The blocks-based environment for this study will be a slightly modified version of the snap! programming language (figure 2). Snap! is blocks-based programming environment that demonstrates all the major features of modern blocks-based environments. The modifications for the most part will not be visible to the user but instead be additional logging capabilities for later analysis (these features will be discussed more later the Methods and Data Collection sections of this proposal). As discussed above, blocks-based programming environments include a number of features to scaffold novice programmers, mostly notably, the defining feature of being able to compose programs by dragging blocks from the organized categories in the Palette onto the Scripting Area and snapping them together to form programs. The blocks in snap! also provide visual cues to facility the learner during the programming activity. The geometric shape of the blocks denote how and where they can be used. Further, the dynamic snapping of blocks together during the construction phase provides feedback to the programmers as to whether or not their most recent addition forms a valid construction.

The Text-based Environment

The text-based programming environment (figure 3) will incorporate a conventional text-based programming interface into the snap! interface. The text-based interface will be overlaid on top of snap!'s Scripting Area, but still display the Stage and Sprite Corral to retain many characteristics of the snap! interface and keep it consistent with the other two environments. Like the other two environments, the programs will control one or more sprites as they move around the stage area. In this environment learners will use the keyboard to type in the instructions of their program and like modern programming tools, the environment will include syntax highlighting, dynamic error checking, and bracket matching visual cues to aid in the construction of programs. The text-based language will be JavaScript. There are two reasons for this: first, JavaScript as a language can recreate all features of snap! including dynamic typing, first-class functions, and various programming paradigms including functional and object-oriented programming. Added to the programming language will be a set of functions that match the available set of blocks, for example, where this is a move 10 steps block in the graphical condition, there will be a `move(10)` function in JavaScript that will produce the same effect.

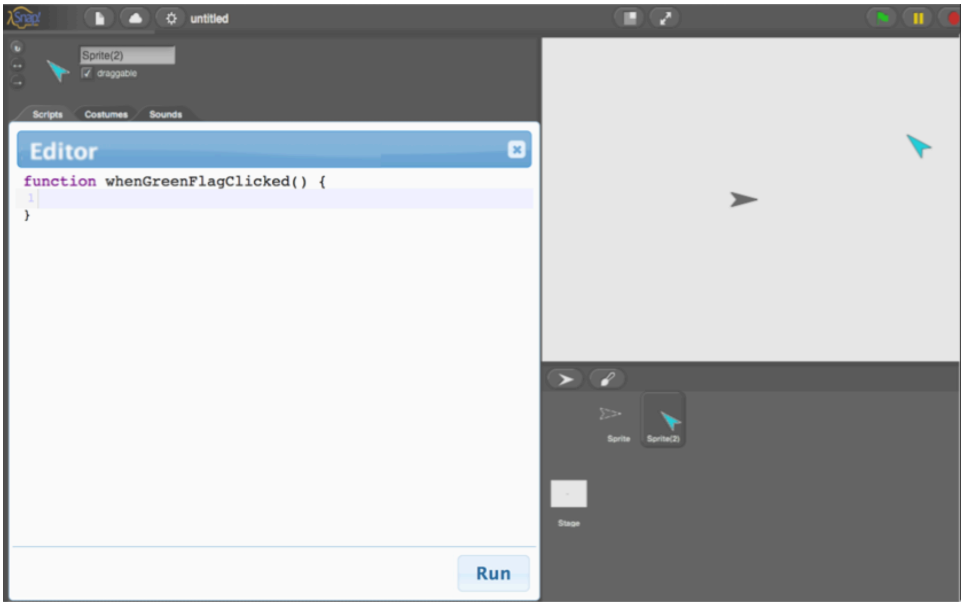


Figure 3. A mock-up of the text-only programming environment.

The Hybrid Blocks/Text Environment

The hybrid blocks/text programming environment (Figure 4) will blend features of both the two prior programming environments. In this environment, users will be able to compose behaviors for their sprites by dragging commands in from the Palette, but unlike the graphical condition, when the blocks are placed, it will be as text in a text editor. Users will be able to browse the language categories, as with blocks and move/drag statements like with blocks, but the entities they are viewing/grabbing/moving will be text. Each event will be a new text area (similar to a window in an operating system). As each of these windows will be text, users can edit and compose as if they were conventional text editors. An added feature to the text interface will be grab-able "handles" that appear next to each text programming statement that can be grabbed and dragged within the text programming area. This will allow the user to re-arranged the code without needing to cut/paste or retype any text.

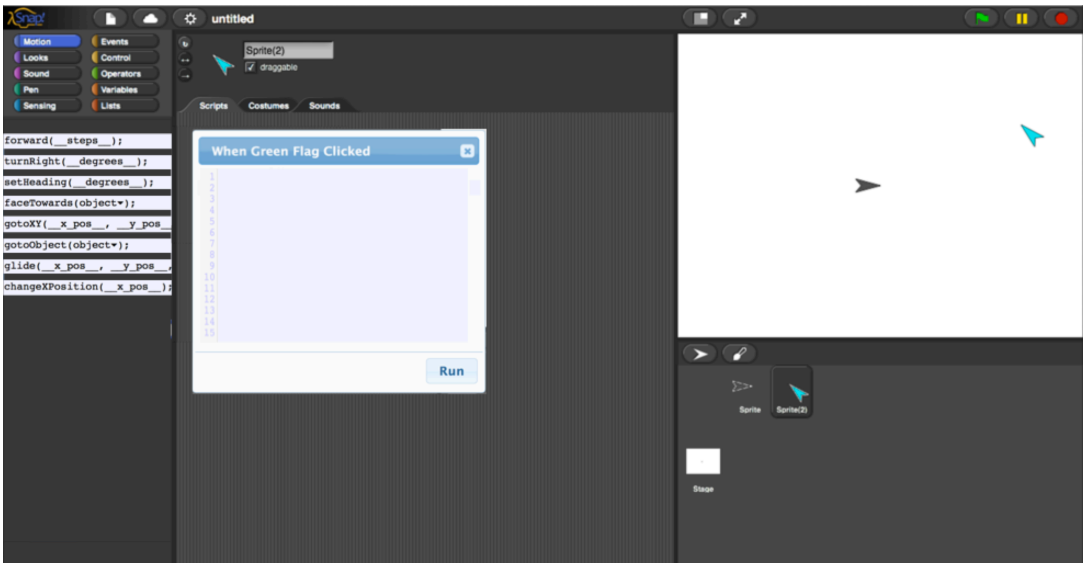


Figure 4. A mock-up of our hybrid blocks-text environment.

Study Design

The study that makes up the heart of this dissertation is a three-condition, quasi-experimental study designed to understand the effects of using blocks-based, text-based, and hybrid blocks/text programming tools in formal introductory computer science classrooms. The study in broken up into two phases: *A Three-way Introduction to Programming* then *The To-text Transition.* During the three-way introduction, we will follow three sections of an introductory programming class at the high school level. Each class will use a different introductory tool for the first five weeks of the school year. The *To-text Transition* will following those same three classrooms as they transition from the introductory tools to more a conventional text-based programming environment, checking in with them throughout the year to evaluate any lasting effects of the introduction. This second phase commences immediately following the conclusion of the three-way introduction phase and will last the duration of the school year. The study will consist of classroom observations, student and teacher interviews, and collecting student-generated materials. We will discuss the data collection methods in more detail later in the proposal.

This section of the proposal begins with an introduction and discussion of the research questions. From there we describe the two phases of the study. For each phase we outline the objective, the intended outcomes (be they artifacts produced or data collected) and provide a description of the procedures that will be followed to achieve those goals. We conclude this section with a description of the setting in which the study will take place and the population that will participate in this study.

Research Questions

As stated earlier, this dissertation seeks to answer three sets of interrelated research questions. The first two are empirical questions on the relationship between the representations used to introduce learners to programming and the conceptual understandings, programming

practices, and attitudes and confidence they engender. The third is a design question exploring and evaluating designs that blend the two modalities. The three sets of research questions are:

1. For text-based, blocks-based, and hybrid blocks/text programming tools, what is the relationship between the programming modality used and learners' understandings of programming concepts? What programming practices do learners develop when working in each of these three environments? How does the representational infrastructure used affect students' perceptions of programming with respect to utility, authenticity, enjoyment, and overall attitudes? And for each of these questions, how do the answers differ across blocks-based, text-based, and hybrid blocks/text environments?
2. How do understandings and practices developed while working in introductory programming tools support or hinder the transition to conventional text-based programming languages? How is this different between text-based, blocks-based and hybrid blocks/text introductory environments? How do learners' understanding of and attitudes towards programming change as learners shift from introductory tools to more widely used, professional programming languages?
3. Can we design hybrid introductory programming environments that blend features of blocks-based and text-based programming that effectively introduce novices to programming and computer science more broadly? How does such an environment perform relative to blocks-based and text-based programming tools with respect to conceptual understanding, development of productive programming practices, and attitudinal, motivational, and engagement outcomes for learners?

Our first set of research questions explores what effects a blocks-based, text-based and hybrid blocks/text representational systems have on a learner while he or she is learning with it. This includes how it influences learners emerging conceptual understanding, the programming practices that develop with the tool, and how the tool affect learners' attitudes and perceptions. Each of these questions will be analyzed using isomorphic blocks-based, text-based, and hybrid blocks/text tools. The second set of questions explores the suitability of each of these modalities used in an introductory programming context for preparing learners for future, text-based programming learning experiences. Answering these questions involves looking at students' perceptions of the representation when used in introductory programming environments and follows them as they move from them to conventional text-based languages. Like with the first set of questions, we will answer these questions by having students work in blocks-based, text-based, and hybrid blocks/text introductory tools. The final research question is a design question intended to explore ways to draw on the strengths of both blocks-based and text-based programming to see if it is possible to effectively create tools that blend the two modalities.

Two Iterations of a Design-based Research Study

Before describing the details of the study we intend to conduct, we want to first describe the larger methodological approach that informs the design of the study. Like many studies in the

field of the Learning Sciences, this study will use a design-based research methodology (Design-based Research Collective, 2003; Collins, Joseph, & Bielaczyc, 2004,). In design-based research, designed artifacts are used to inform our understanding and advance our theory for how students are coming to understand the topic under investigation. In our study, the three variations of our introductory programming tool serve as the central design component with which we will investigate student understanding. A central characteristic of design-based research studies is their iterative nature, where earlier trials with a given tool are used to inform and revise the designed artifacts. This study will conduct two iterations in two consecutive school years. The first year will be used as a pilots study for the programming environments, the curriculum, and the data collection methodologies of the study. The data collected in the first year will be used to inform and revise the materials and procedures the following year. These revisions will be based on our observations of how the tools and materials worked in the classroom, feedback from students and teachers, as well as based on our preliminary findings from analyzing the data collected during the first year of the study.

Phase One: A Three-way Introduction to Programming

The first phase of the study will bring the three introductory programming environments into high school classrooms. This phase of the study will follow three classrooms during their first 5 weeks of a year-long introduction to programming course. Each of the three classes will use a different environment, either a fully graphical environment, a text-based environment, or a hybrid blocks-text programming environment. All three classes will work through the same set of activities and engage in the same classroom discussions. We recognize that the pacing of the classes may differ based on the tools used, so while all three classes will start on the same lesson, one class may advance through the curriculum at a faster rate, that will result in more time given to the summative project. We are not controlling for this pacing as the speed with which students become comfortable in an environment and the ease with which a teacher and introduce material using that tool contribute to the overall effectiveness of the tool that we are interested in evaluating. Our hope is that a single teacher will teach all three sections providing further continuity across the conditions, but we recognize that this decision is in the hands of the department chair of the school so it is more likely we have two teachers teaching the three class (as is the case for the first year of the study). In this case, we will have one teacher teach the text-based and graphical conditions, and the second teacher teach the hybrid condition.

Curriculum

The five-week curriculum for the introductory course is based on the Beauty and Joy of Computing course designed by Daniel Garcia and Brian Harvey at UC Berkeley. This course is intended for non-computing majors and is designed for the snap! programming environment. We spend the first four weeks moving through different computing concepts and conclude the 5-week curriculum with a summative project that gives students the freedom to create the game, interactive story, or other project of their choosing. Table 2 provides a high-level outline of the curriculum.

Table 2. A high-level summary of the 5-week introductory curriculum.

Week 1: Introduction to snap! The goal of this first week is to acclimate students to that act of programming in snap! This includes introducing them to the concept of a sprite, how blocks/commands control the sprites actions, how users interact with programs (keyboard/mouse inputs), and features of programming interface (the Stage, Palette, Scripting Area, etc.). **Activities:** Bounce off edges, Follow the Perimeter, Follow the Mouse, Kaleidoscope
Week 2: Abstraction and Custom Blocks/functions In this second week we have students begin to create their own new custom blocks which includes ideas of abstraction, functions, and parameterization **Activities:** Polygons, Flower Garden
Week 3: Iterative Logic and Variables In week three, we introduce looping logic. The forever block is the first to be introduces as it is often used in animations. We then introduce the repeat and while blocks. **Activities:** Random Triangles, Brick Wall
Week 4: Conditional Logic The fourth week introduces conditional logic and predicates. **Activities:** Conditional Zombies (blend of conditional and iterative loops), Leap Year, Mystery Number
Week 5: Summative project The final week of the curriculum has students develop their own projects. The only requirement is that projects must include all of the concepts students have encountered (custom blocks, variables, iterative logic and conditional statements). Students spend four days working on their projects then present them to their peers on the final day of the initial phase of the study.

Phase Two: The To-Text Transition

The second phase of this study seeks to shed light on the question of how well each of the three introductory tools used in phase one prepare students for future computer science instruction in more conventional text-based languages. In this phase we follow students as they transition from the introductory programming environments used in the first five-weeks of class to more conventional text-based programming languages and environments that serve as the basis for future computer science instruction. All students will use the same language and programming tools in this phase, regardless of what condition they were in during the first phase of the study. In this phase, students will learn the Java programming language. The programming tools will be decided by the teachers who are teaching the course as these tools will be used for the duration of the year. In the first year of the study, one teacher chose to use BlueJ, a popular Java programming tool designed for beginners, while the second teacher had students program using a text editor and compile and run their programs on the command line.

We will follow students through their first three weeks in Java and then return to the class for three, weeklong observation and interview periods. These three later visits will occur during

the first week of the three curricular units: data types, conditional logic, and iterative logic (in that order). The reason for returning to the classroom at regular intervals is two fold. First, this will give us insight into the lasting effect the initial 5-week intervention had. Do students still reference the environments used at the start of the course 10 weeks later? How about 15 weeks? The second reason for revisiting the classroom is based on the year-long design of the course. The course follows the Java Concepts: Early Objects text book (Horstmann, 2012) which, as the name suggests, uses an objects-first approach. This means students encounter the concepts of objects and classes before conditional logic and loops. While objects are indirectly encountered in our introductory curriculum (via the sprite) they are not the focus of the introductory activities, so we do not expect to see students draw on the introductory environments as there is not as direct a conceptual link.

Setting and Participants

This study will take place at a public high school in a Midwestern city that has an established computer science program. The experimental materials will be incorporated into an existing introductory programming course that in past years has used Scratch for the first 3 weeks of the course to introduce learners to programming concepts before transition to Java for the remained of the year. The teachers who teach this course all have at least 5 years of computer science teaching experience at the high school level. The introductory programming classrooms the study will take place in will each have roughly 30 students and include students across all four years of school. The computer science course is an elective but historically has attracted students from a variety of racial background and been taken by both male and female students. The school is 47% Hispanic, 31% White, 10% Asian, and 8.5% Black with 58% of the students receiving free or reduced price lunches. The administration in this school has also shown a great deal of support both for computer science education and for pursuing innovative educational programs making it especially well suited as a research site for this work. One draw back of this study setting is that it is a selective enrollment school and consistently ranks as one of the best public high schools in the city, meaning the students who participate in the study are not necessarily representative of the larger city-wide student population.

Conceptual Framework

Having reviewed prior work that informs the design of both the environments we are building and the study we are conducting in the literature review section, and laying our the design of the proposed study above, we now present the conceptual framework that we will use to answer our stated research questions. After presenting the conceptual framework we will conclude this proposal by outlining the methods and data collection strategies we will follow, and laying out a timeline for this work to be conducted.

Webbing and Situated Abstractions

Our analysis of learners making meaning of computational ideas will build on work by researchers pursuing similar goals in the domain of mathematics education and a pair of theoretical constructs they developed to answer such questions. In their work on the construction

of mathematical meaning in computational environments, Noss and Hoyles (1996) pursued two research questions that are closely aligned with the questions pursued in this dissertation. Their first question asked how resources of computational environments supported students' mathematical activities, for which the construct of webbing was developed. Their second question was to gain insight into the nature of the mathematical abstractions within computational settings, for which the construct of situated abstractions was developed. In this dissertation, the domain of interest is computer science as opposed to mathematics, but the underlying questions, examining the relationship between computational environments and the meanings created through their use, is shared. This similarity in purpose motivates our intended use of the constructs of webbing and situated abstraction in the analysis of the data we will collect to shed light on our research questions. In this section, these two constructs and the ideas on which they are built and the frameworks upon which they rest are discussed as they form the basis for the conceptual framework we bring to this work.

Vygotsky, as part of this theory on the sociocultural perspective of cognition developed the idea of the zone of proximal development, defining it as the "distance between the actual development level as determined by independent problem-solving and the level of potential development as determined through problem-solving under adult guidance or in collaboration with more capable peers" (Vygotsky, 1978, p. 86). This realization of the positive role that external actors and artifacts can play in support of a learner's development lead to the development of the notion of scaffolding, which began as a term for describing supports provided to a learner by an expert (Wood, Bruner, & Ross, 1976), but has since been broadly expanded to include a wide variety of artifacts designed to support learning (for a review of different usages of scaffolding, see Sherin, Reiser, & Edelson, 2004). In assessing how well the notion of scaffolding translated to the constructionist computer-based learning environments they were investigating, Noss & Hoyles (1996) identified enough differences that they deemed a new construct was necessary. Noss & Hoyles, drawing on the work of Wilensky (1991), view meaning-making as a relationship-building process during which learners forge connections and build relationships with a concept as they come to understand it, a process Wilensky called concretion. Missing from the notion of scaffolding was the ability to account for the multifaceted, situated, and personal nature of this process of concretion. The idea of webbing was thus introduced to describe "a structure that learners can draw upon *and reconstruct* for support – in ways that they choose as appropriate for their struggle to construct meaning" (p. 108, emphasis in original).

A number of distinctions were made differentiating webbing from scaffolding that make it an appropriate lens for answering the questions posed in this dissertation. First, webbing captures the full network of supports provided to the learner, not just a single feature of the environment. By recognizing the webbing includes all these components we are able to talk about features of the environment used in concert, as opposed to elements in isolation, which more accurately captures how the interviews were conducted. A result of this shift in focus away from a single specific feature is that it enables the learner to remain in control of his or her

learning progression, as opposed to following an externally prescribed path dictated by a particular scaffold. A second difference between scaffolding and webbing is that scaffolding is designed to fade as expertise grows. A final distinction is that webbing is recognized as not being universal across settings and learners but instead "domain contingent…it focuses attention on the influence of the setting and the symbol system within which the ideas are expressed" (Noss & Hoyles, 1996, p. 109). This feature of webbing allows our analysis to be consistent with the constructionist principles upon which the environment is built as it recognizes the personal nature of the meaning making process.

Using the construct of webbing, we can talk about aspects of the three designed learning environments that support learners in their personal learning voyage without needing to be concerned with aspects of the notion of scaffolding that do not fit our tools. The next logical step for our analysis is figuring out how to work from the webbing of a specific observed event and the personal nature of the concretion process, back to the general concepts and practices of the domain of interest, in this case, computer science. For this, Noss and Hoyles (1996) develop the notion of situated abstraction to describe "how learners construct mathematical ideas by drawing on the webbing of a particular setting" (p. 122). As such the notion of situated abstraction was developed in order to "afford a means to describe and validate an activity from a mathematical vantage point, without *necessarily* mapping it onto standard mathematical discourse" (Hoyles & Noss, 2004, p. 2, emphasis in original). In their interactions with these computational environments "learners web their own knowledge and understandings by actions within the microworld, and simultaneously articulate and mesh fragments of that knowledge – abstracting within, not away from, the situation" (Noss, Healy, & Hoyles, 1997, p. 228). The construct of situated abstraction gives us a way to both attend to the situated nature of the act of learning within the webbing of a computational environment, while also recognizing the ability of concepts and practices to transcend contexts, and thus providing a way to link in situ activity with the more general, abstract forms of conceptual knowledge from the domain of interest. We have followed this approach in previous work and found the use of these two constructs productive for elucidating the emerging understandings the learners develop working in low-threshold programming environments (Weintrop & Wilensky, 2014).

Interpretive Devices and Symbolic Forms

The constructs of webbing and situated abstraction give us a way to view the relationship between the tools used and emerging understandings and direct our attention towards certain features of the learning context. To inform our analysis at a more fine-grained, microgentic level, we draw on the work of Sherin, whose work on students' understanding of physics concepts based on differing representational tools closely mirrors our own goals. In his work trying to understand the affordances of representations towards supporting different cognitive strategies and thus, different conceptual tools, Sherin (2001a, 2001b) developed a pair of analytic tools: *interpretive devices* and *symbolic forms*.

A feature of programming languages that differentiates them from other static representations is that they can be brought to life; they can be run (diSessa, 2000). This feature

grants the representation a dynamism in the virtual world that can be replicated internally; programmatic representations can be run mentally, a feature learners leverage as they work to comprehend their meaning. The structures learners develop to facilitate this strategy Sherin calls interpretive devices. The development of these devices is coupled with learners developing a conceptual vocabulary of symbolic features of the representations used. This set of symbolic features of the representation in turn informs how learners interpret and develop a conceptual understanding the ideas encoded in the representation. Called *symbolic forms*, these are visual queues learners become attuned to and recognize as encoding a given concept. Where Sherin compared algebraic expressions and text-based programmatic statements, we undertake a comparable analysis of text-based and blocks-based programming languages. We see this as a promising avenue as this type of analysis will allow use to type specific interpretive strategies with features of the representation, and given it is the visual presentation of the two modalities that is of interest in this dissertation, the approach seems especially well suited.

These analytic tools complement webbing and situated abstractions discussed above. Where Noss and Hoyles' situated abstraction provides a means to capture an emerging understanding in-the-moment with a specific learner and tool, interpretive devices and symbolic forms provide a way to identify and map out the set of features of a representational system that contribute to the webbing from which this meaning emerges. Taken together we can begin to identify the role of certain features of a each modality that influence the conceptual understandings and programming practices that emerge in learners through their use of the tools.

Methods and Data Collection

This study will utilize a mixed-method approach to answer the stated research questions. This will include quantitative and qualitative methodologies as well as computational methods to analyze the large dataset of learner-authored materials. In this section we breakdown each component of our data collection strategy including what will be collected, how it will be gather, and how we intending on using that data to answer our stated research questions. We begin by discussion our quantitative data sources then continue with our qualitative data, then conclude the section by discussion the computational methods we intend on employing.

Quantitative Data Sources

The main quantitative data sources for this study will be a series of attitudinal surveys and content assessments. The surveys will be administered four times over the course of the study: (1) at the outset in week 1, (2) between phase 1 and phase 2 which is after students have completed the portion of the study where they will be using the introductory programming environments, (3) after the conclusion of the first Java unit, this will be roughly 10 weeks into the course, and (4) at the beginning of the iterative logic unit which will be the last time I will be in the classroom and occur roughly 20 weeks into the course. Each administering of the surveys helps us answer a different question. The initial set of data gives us a base line for each student and the classes as a whole. The second set will allow us to measure the impact of the introductory programming environments, both within each condition to see who effect each tool

was as well as across conditions to compare how well they do to each other. The third data set will allow us to measure students' initial experiences learning to program in Java. This data set will give us insight into if students' attitudes or confidence have shifted since moving to text-based programming as well as information about if and how concepts learned with the introductory tools are still salient after leaving those tools behind. The data collected in the fourth and final administration will provide a summative data point for students attitudes toward programming and perceptions of the field of computer science. With this last data set, students will have been working in Java for almost 15 weeks, giving them a more informed perspective of conventional programming and will be relatively far removed from the introductory environments designed to engage learners and get them excited about the field. Administering these two instruments multiple times over the course of the study will give us power to speak to the effects of the tools both immediately as well as their lasting effects.

The surveys will be administered online during class time on consecutive days so as to minimize testing fatigue. We expect that attitudinal survey will take students around 15 minutes and the content assessment to take close to 25 minutes. We will record timestamps with each submission so will know how longs students spend on each assessment. The surveys will be hosted on Google Forms and the responses will be recorded in a Google spreadsheet. Students who are absent the day it is given will be asked to take fill out the survey outside of class time.

Attitudinal Surveys

The attitudinal surveys are loosely based on materials used as part of the Georgia Computes initiative (Bruckman et al., 2009). The questions in this survey are designed to get insight into a number of attitudinal and perceptual facets. The survey begins with a number of Likert scale (1-10) questions asking students about their confidence in taking the course, their excitement for the course, and their general perception of the field of computer science. The survey then continues with short answer questions about motivations for taking the course, prior experience with programming and computer science culture, things they are excited to learn as part of the class, and some open-ended prompts about how the view the field of computer science. The survey questions will remain largely the same over the course of the study, although tenses will change to align with the time the test is being administered (from future, to present, to past tense). Also, a few questions will be added/removed to fit the point in time when the survey is given. A full copy of the attitudinal survey can be found in Appendix 1.

Content Assessments

While a standard concept inventory for introductory computer science has yet to be established (Taylor et al., 2014), there are a set of concepts that are recognized as being central for such courses. We primarily draw on two resources in deciding what concepts to include in our assessments. The first is the recently released 2013 CS Curriculum (ACM/IEEE-CS Joint Task Force on Computing Curricula, 2013) that is meant to provide guidelines for university computer science departments. This curriculum breaks down the field of computer science into broad categories and recommends how much time should be committed to each category and at

what point the material should be covered. In reviewing this document, we selected concepts that were emphasized as being foundational early in a students' career. The second resource we draw on is the work of Tew and Guzdial (2010, 2011) and their effort to create a validated CS 1 assessment. As part of this effort, they reviewed the contents of 12 introductory computer science textbooks along with other published curricula to establish a list of core CS1 concepts. Their final list consists of:

- Fundamentals (variables, assignment, etc.)
- Logical Operators
- Selection Statements (if/else)
- Definite Loops (for)
- Indefinite Loops (while)
- Arrays
- Function/method parameters
- Function/method return values
- Recursion
- Object-oriented Basics (class definition, method calls)

Informed by these two resources, we decided to focus our content assessment on the subset of these concepts that could fit within our five weeks adaptation of the Beauty and Joy of Computing curricula. This included the following five categories from Tew and Guzdial's work: fundamentals, selection statements, definite loops, indefinite loops, and function/method parameters (note: we condensed definite and indefinite loops as they were taught together in our curriculum). Based on our review of the CS2013 Curriculum and what it emphasizes for introductory courses, as well as our desire to broaden the assessment beyond programming specifics, we decided to add two additional content categories: program comprehension (interpreting the behavior of short programs), and algorithms (natural language descriptions of steps to be followed to solve a problem). Table 3 shows our final list of concepts and how it maps on to Tew & Guzdial (2010) and the CS2013 curriculum.

Table 3. The computer science concepts covered in our content assessment.

CS 1 Concept Assessed	Mapping to Tew & Guzdial (2010) Categorization	Mapping to CS 2013 Curriculum Category
Variables	Fundamentals	Fundamental Programming Concepts
Conditional Logic	Selection Statements	Fundamental Programming Concepts
Iterative Logic	Definite Loops; Indefinite Loops	Fundamental Programming Concepts
Functions	Function/method parameters; Function/method return values	Fundamental Programming Concepts
Program Comprehension	-	Development Methods

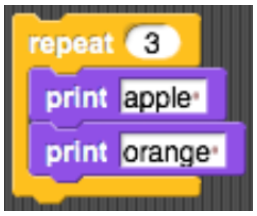
Algorithms	-	Algorithms and Design

Each assessment asks 5 questions for each of the 6 categories for a total of 30 questions. All of the questions will be multiple choice or true/false and, with the exception of the algorithm questions, all will take the form a short piece of code that students are asked to state the output. The algorithm questions will be text descriptions of a problem then ask the students questions about steps that need to be taken to solve that problem. The multiple choice answers will include the correct answer along with options that would be the outcome if students hold a misconception that has been identified in the literature (see appendix A of (Sorva, 2012) for a summary of student misconceptions of programming concepts). For example, Figure 5 shows a sample iterative logic question from the assessment. Students will see either the text version or the blocks version, then choose what they think will be output by the program.

What will be printed when this program is run?

```
for (var c = 0; c < 3; c++) {
  print("apple ");
  print("orange ");
}
```

(or)



A) “apple apple apple orange orange orange ”
B) “apple orange ”
C) “apple orange apple orange apple orange ”
D) “” [nothing with be printed]

Figure 5. A sample iterative logic question.

The choices presented are drawn from misconceptions literature on how to understand conditional logic. Choice C is the correct answer as the output correctly alternates between the two printed statements. Choice B would be chosen by a student who does not know that the repeat block or for loop cause the commands nested inside its scope to execute multiple times. Option A comes from a misconception identified in the literature, that students think statements within a loop execute in the order they are shown. Including responses drawn from the misconceptions literature will help provide evidence for linking certain modalities with misconceptions about the concepts that are being demonstrated. The full set of questions used in the content assessment can be found in Appendix 2.

The code used for the questions in the content assessment will all have the feature that it can be expressed either in a blocks-based or text-based manner. For each administration of the content assessment, half of the questions will be presented with blocks-based code and the other half will use text based. All four content assessments will ask the same set of questions, but the order and the modality (blocks vs. text) will change. The questions in the second content assessment will use the opposite modality from the first assessment, so after taking the content

assessment twice, all students will have seen every question in both modalities. For the third and fourth assessment, two version of the assessment will be used. They will follow the same order but present questions in opposite modalities. The goal of this is to identify how well students understand and interpret questions for different concepts based on the modality presented.

The design of our content assessments will make it possible to group the responses along a number of dimensions that will collectively yield insight into the relationship between modality, tools, and emerging understanding. The dimensions include grouping responses by: condition (what tools the students used), representation used in the question (graphical vs. textual), concept (conditional logic, iterative logic, etc.), prior experience (students who have taken Exploring Computer Science vs. those who have not), and various combinations of those groupings. This approach will provide data to support or refute claims about whether one modality is easier to interpret than another with respect to the various concepts included in our written content assessments as well as link the introductory tools used with fluency with concepts across different modalities.

Qualitative Data Sources

A number of qualitative data sources will be gathered as part of this study to compliment the quantitative data just discussed. These data sources include: classroom observations and recordings, informal student interviews during class, semi-structured student and teacher interviews outside of class, and the collection of non-computational student generated artifacts.

Over the course of the study, the classroom will be observed regularly. Each observation will include field notes of what is happening in the classroom as well as video recording of the teacher when he or she is presenting to the class and audio recordings during classroom discussions. Classroom observations will be focused on student questions, teacher strategies, and characteristics of the class dynamic that are influenced by the programming environment. As is common in programming courses, much of the class time will be devoted to students working on assignments. During these portions of the class there is generally little classroom discussion or formal teacher instruction. When students are working independently, researchers will walk the room and conduct brief interviews with students, recording their screens while the student explains the program they are authoring. Researcher questions will attempt to get students to explain why those chose to compose their program a certain way and how behavior would be different if various changes were made. Also during this period, the researcher will assist students with questions, similarly recording the students screen during these conversations. This data source will help to elucidate difficulties and misconceptions students have about different programming concepts.

We will also conduct student and teacher interviews outside of class over the course of the study. Student interviews will have a researcher sitting alongside a student either asking the student questions about their experiences in the class or have them think-aloud as they work activities designed to illicit specific types of thinking around computer science concepts. The goal of these interviews is to more deeply probe students emerging understandings and identity if and how understandings are bound up with the representations they are using the classroom.

Four waves of interviews with students will be conducted. Each set of interviews will include students from all three conditions of the study and follow a separate protocol. The interview protocols can be found in Appendix 3. The first wave will happen at the start of the course, the second at the conclusion of the introductory tools portion of the study, the third will be after students have been using Java for roughly 5 weeks, and the last wave will occur roughly 20 weeks into the course, after students have been programming in Java for 15 weeks. These interviews will be recorded with screen capture software and the computer's on-board camera and microphone and serve as the primary data source for understanding the relationship between the representation used and students' emerging understandings.

Teacher interviews will be conducted twice during the first five weeks of the study and then once each week we are back in the classroom observing. These interviews will be conducted outside of class either after school or during the teachers' preparation periods. These interviews will ask the teacher to reflect on the tools being used in this class and compare it to his or her experiences using other introductory tools. We will also discuss specific assignments and students in each of the classes. The goal of these interviews is to draw on the teachers' expertise to gain insight into how they perceive the students are reacting to the different tools. Additionally, we are interested in how well the various tools perform in a formal classroom setting, so these interviews will provide an opportunity to gather data on these questions. The teacher interviews, unlike the student interviews, will be less formally structured and be more of a conversation about the classroom and their experiences teaching with the tools. We take this approach because we want to draw on the knowledge and experiences of the teacher and not keep them to a specific script for fear of missing important classroom events.

The final form of qualitative data we will collect will be non-computational student created artifacts. In particular, one teacher who we are partnering with has students keep a journal over the course of the school year. In these journals the teacher has students write pseudo-code, respond to class prompts about different computer science concepts and reflect on in-class activities. We will use these journals to look for further evidence on the relationship between how students understand concepts and how they compose solutions (in the form of pseudo-code) and the programming modality they have are using in class. We plan on having this teacher's classes use the text-based condition and the blocks-based condition, providing the further contrast between her two classes, giving us the ability to use the journals to compare the two modalities directly.

Computational Data Sources

The third component of our data collection plan will involve collecting snapshots of the programs students write throughout their time participating in the 10-week study. In taking this approach we intend on using computational methods to identify trends in how students develop programs and unique characteristics of the programs they produce and see if and how they differ across students from the three conditions. This approach builds on a growing body of research looking at the use of big data for studying student learning (Baker & Yacef, 2009). This field has been especially active in the domain of computer science education as students incrementally

building programs lends itself well to the gathering time series data and the relative uniformity across final projects makes it possible to computational compare final projects (Jadud & Henriksen, 2009). Recent work has looked at trends in how students write programs (specifically looking at frequency and size of incremental updates), intermediate program states the students visit as they work toward the final project, and the predictive nature of programming features on students' final course grades (Berland, Martin, Benton, Petrick Smith, & Davis, 2013; P Blikstein et al., 2014; Werner, McDowell, & Denner, 2013). Our strategy for collecting and analyzing student projects follows this line of work.

Conducting this analysis depends on the ability to record every iteration of every program written by students participating in this study. Over the course of the 10-weeks, we will record a snapshot of a student's program each time they compile and/or run it. As two different languages are being studied (snappier! and Java), two different sets of data collection tools will be developed. In this section we describe our strategy for collecting student programs in each environment, then discuss our larger data analysis strategy.

Snappier! Data Collection Tools

All three versions of our introductory tools that built on top of the snap! programming tool will use the same data collection system. Added to the snap! environment is logic that will record information about the current state of the programming environment and send it, along with information that allows us to identify the author, to a server that will then record that information in a database. Before students start using the environment they will have to log in, this will allow us to link any recorded program with the student (or students) who authored it. In all three environments, we will record the program the student has written every time the program is run and at the conclusion of every class when the programming environment is closed. We consider a program 'run' to be any time a script is executed, usually resulting in a sprite on the stage taking action, but not always. In snap! there are three ways a script can be run. First, a user can click on a block or script in the Scripting Area. Second, a script can be set to run when the Green Flag icon is clicked. Finally, scripts can be associated with inputs, such as a specific key press. All three of these interactions will be recorded along with an identifier so it is possible to link any given run to the action that initiated it.

Each time a program is run we will record a full copy of the program. This will be done by issuing an asynchronous HTTP request in the background of the programming environment (the browser) that will post the contents of the composed program along with other pertinent information to a server that we control. Each request will be stored in a database on the server that will create a record that includes the program's content, the user who authored it, the time it was received, and the environment that issued the call (blocks-based, text-based, or hybrid). In the graphical environment, this will be done using a modified version of snap!'s exporting feature that encodes the blocks, their placement and argument values as xml. In the hybrid and text-based programming conditions, every time a program is run we will log the entirety of the text that has been written. Table 4 outlines what information will be recorded for every snap! program run that occurs.

Table 4. Data recorded via our automated data collection tool in the three introductory tools.

Column Name	Description of data being stored
`StudentID`	A unique identifier for the primary author of the program
`PairID`	In the case when students are pairing on a program, this field records the unique identifier of the second student working on the program (optional)
`ProjectName`	The student defined name of the current project
`TimeStamp`	The time (to the millisecond) that the logging request was recorded
`RunType`	The type of run event that triggered the logger; valid entries include: threadStart (script clicked directly), GreenFlag, ProjectClose
`Condition`	The condition of the student the student is in (graphical, textual, hybrid)
`ScriptXML`	The contents of the program
`NumRuns`	The number of times that scripts has been run

Java Data Collection Tools

Just as we collect each program written by students working in the three versions of our introductory tool, we also plan to collect all programs written by students once they have transitioned to programming in Java. Like with the introductory tools, our intention is to log a copy of every program that students run (or attempt to run). In the case of Java, this means every time students compile a program, we will log the compilation command, the content of the programs being compiled, and the output of the compilation. Our partner teachers use different Java programming environments, which means two Java logging solutions must be developed. Both logging tools will be designed to run in the background on the students' computers so they are not aware of the logging that takes place. This is so that the logging process does not interfere with classroom practice or the students learning. Student programs will be stored on a remote web-server that will for incoming requests from the two different Java programming environments.

Command-line Data Collection with JavaSeer

The first Java data collection approach will be used in classrooms where students compile and run their program from the command line using the javac command. To record these compilation events we developed a tool we call JavaSeer that will replace the students' javac command with a script of our own that will mimic the behavior of the javac command from the student's perspective but give us access to the student programs for logging purposes. The way JavaSeer works is that it creates an alias to the javac command line call within the terminal on the computers the students use. When students run javac, JavaSeer reads in the arguments the student passed to javac, which includes the list of files the student intends on compiling. Inside JavaSeer we call javac with the same commands the student passed in, we then record the output from javac and pass it back to the user as output. Additionally, JavaSeer reads in the contents of the files being compiled and send them, along with the compilation output and other information to the JavaSeer server via an HTTP request. This whole process is invisible to the student. This

approach to logging student programs was informed by the Git Data Collection project and the work built as part of that effort (Danielak, 2014). Table 5 describes the data that we collect for each program via JavaSeer.

Table 5. Data recorded via JavaSeer, our automated data collection tool for command line compilation.

Column Name	Description of data being stored
`StudentID`	A unique identifier for the primary author of the program
`JavacCall`	the argument passed to javac (which will be the list of file names)
`TimeStamp`	The time (to the millisecond) that the logging request was recorded
`JavaProgram`	The contents of the java files that are being complied
`JavaCompilerOutput`	The compiler output from the call to javac

BlueJ Data Collection with BlueJChirp

The second Java data collection approach takes the form of an extension to the BlueJ programming environment. BlueJ is a widely used Java programming tool for novices that has features supporting graphical layout of files within a program, additional supports for interacting with running programs, improved error messages, and visual cues to depict various program structures (Kölling et al., 2003). One of the teachers we are partnering with uses BlueJ in his classes and, as such, we had to devise a strategy for logging student programs for students in this condition. We accomplished this by building off of the work of Jadud (2005) who pioneered the logging of student programming artifacts as part of his dissertation, which was written while a member of the BlueJ research group. BlueJ provides an extensions API to interact with and add functionality to the environment. Following the work of Jadud and his BlueJ Logger project, we wrote a new extension for BlueJ, called BlueJChirp, that was added to the BlueJ instances of students in the classroom. This extension records the same set of information as JavaSeer and pushes it to the same remote server that JavaSeer calls to, but stores the data in a separate table (due to some sleight difference in data and format). Table 6 describes the data that we collect for each program via BlueJChirp.

Table 6. Data recorded via BlueJChirp, our automated data collection tool for BlueJ.

Column Name	Description of data being stored
`StudentID`	A unique identifier for the primary author of the program
`JavacCall`	The name of the file being compiled
`TimeStamp`	The time (to the millisecond) that the logging request was recorded
`JavaProgram`	The contents of the java files that are being complied
`JavaCompilerOutput`	The compiler output from the call to javac
`NumCompiles`	The number of times that file has been compiled

Differences between JavaSeer and BlueJChirp

There are a number of minor differences between JavaSeer and BlueJChirp that are worth mentioning. First, BlueJChirp calls are by class, meaning if the student compiles three classes at once, that will result in three records to the BlueJChirp table, the equivalent call in JavaSeer will produce only a single entry in its table (although that entry will contain data on all three files). Also, BlueJ only reports a single error for a compilation, even if multiple errors are present. Additionally, the text of these errors is different than the standard javac output, in an attempt to make debugging easier on novices. This means the error logs for the two environments will differ. While these differences exist, it will be possible to overcome them in the analysis stage by reformatting data from each database table to be fit the same data structure for analysis.

A Final Note on Java Data Collection

In the first year of our study, our two partner teachers used BlueJ and the command line in their classrooms. At the time of writing, we are under the impression we will work with the same teachers next year, but we have been told by the chair of the computer science department at the school we are working with that this is not a guarantee, and that it is possible that different teachers will teach the introductory programming course next year. If this is the case, and the new teachers use other tools (like eclipse, intelliJ, GreenFoot, or some other Java development environment) we will develop new tools that support collecting the same set of data that we are collecting with JavaSeer and BlueJChirp.

Computational Analysis Methods

Here we briefly describe the approach we intend on taking for analyzing the large number of programs we will collect over the course of the 10-week study. One of the questions we seek to answer is understanding how programming differs between different modalities and if and how practices developed in one modality persist or fade when moving to another. As we will have data on student programming in introductory graphical, hybrid, and text environments along with data from those same student programming in a conventional text-based programming environment, our data set will allow us to begin to answer these questions. The analytic approach we intend on taking with this data is similar to the approach used in Blikstein et al. (2014). Our contribution to this approach is the inclusion of two different programming languages that use different modalities and the ability to look across them and compare the programs created. Further, our quasi-experimental design will give us power that other similar studies have lacked. It is worth noting that while what we state below is our intended plan of work, portions of the analysis will be exploratory, so there are potential to deviate from this course.

The first type of analysis we will conduct will look not at the contents of the programs produced, but patterns in the development practices adopted by learners across the different environments. This includes frequency of compilations, the size of changes made between runs, and the amount of time spent running programs that have been authored. This data will allow us to categorize the programming practices developed in the introductory tools and compare them with the practices used in the conventional Java programming environments. We hypothesize

that in the graphical introductory condition, learners adopt a very rapid development cycle, highlighted by small changes and frequent compilations and runs. In contrast, we expect that students in the text-based introductory tools have longer gaps between runs of their programs and few total runs. We expect this trend to continue in the Java environment, where students spend more time authoring their projects and less time compiling and running their programs. We will include questions about differences in programming between the environments as part of our interview protocols as a second data source to verify the differences that exist between programming in the different modalities.

The second type of analysis we will conduct will look at the contents of the programs themselves to look for trends in use of language features or differences in approaches to program composition. This analysis will be more exploratory than the first as we are not sure what difference (if any) will emerge from each of the three introductory conditions. We intend on selecting a subset of assignments and looking for similarities and differences between solutions authored using the different tools and by students from different conditions (in the case of Java assignments). Part of the goal of this effort is to identify various states that programs can be in during the authoring process and seeing if and how students navigated this space. To do so we will try using a number of standard computational methodologies including a *bag of words* analysis (Salton, Wong, & Yang, 1975), and looking for AST similarities using the Evoluizer algorithm (Gall, Fluri, & Pinzger, 2009). Our hope in using these various approaches to measuring program distance is that we can identify trends that differ among students in the different study conditions, and that we can then tie these trends back to features of the environments that they initial used. If successful, this will give us power to make claims about the effects that introductory programming tools have on the emerging understanding and programming practices of the novices who use them.

Timeline

First Iteration

Activity	Output/Data	Time
Develop preliminary materials	• 3 programming environments • Logging tools • Content Assessment • Attitudinal Assessment • Interview Protocols	Summer 2014
Gain access to research site	Northwestern IRB and CPS RRB approval	Summer 2014
Phase 1 (3-condition introductory tools)	• Record student authored programs • Conduct two rounds of student interviews • Conduct two rounds to teacher interviews • Observe classrooms (~3 days a week)	Sept – Oct 2014
Phase 2 (Java tools)	• Record student authored programs • Conduct two rounds of student interviews	Oct-Nov 2014

	• Conduct two rounds to teacher interviews • Observe classrooms (~3 days a week) • Collect student created materials (journals)	
Data Analysis	• Develop preliminary coding scheme • Develop computational strategies	Winter 2015

Second Iteration

Activity	Output/Data	Time
Develop second iteration of tools	• 3 programming environments	Spring-Summer 2015
Revise data collection tools and protocols	• Logging tools • Content Assessment • Attitudinal Assessment • Interview Protocols	Summer 2015
Gain access to research site	Northwestern IRB and CPS RRB approval	Summer 2015
Phase 1 (3-condition introductory tools)	• Record student authored programs • Conduct two rounds of student interviews • Conduct two rounds to teacher interviews • Observe classrooms (~3 days a week)	Sept – Oct 2015
Phase 2 (Java tools)	• Record student authored programs • Conduct two rounds of student interviews • Conduct two rounds to teacher interviews • Observe classrooms (~3 days a week) • Collect student created materials (journals)	Oct-Jan 2015/16

Analysis, Writing, and Profit
Winter/Spring/Summer 2016

Appendix 1 – Attitudinal Survey

Pre Survey

The following questions gather basic demographic information about the students:

- Name
- Student ID
- Programming Class (dropdown)
- Birthday
- Grade (dropdown)
- Gender (dropdown)
- Race/Ethnicity (checkbox)
- What language (or languages) do you speak at home?

The following questions are 1-10 point Likert scale questions about programming and computer science generally

- Programming is fun
- I will be good at programming
- Programming is hard
- I know more than my friends about programming
- Most women can learn to program
- Programming jobs are boring
- I like programming
- Most men can learn to program
- My family encourages me to learn to program
- My friends like using computers
- I can become good at programming
- I like the challenge of programming
- I think programming is useful
- I am interested in a career in programming
- Knowing how to program is important
- I cannot learn to program well if the teacher does not explain things well
- I plan to take more computer science courses after this one

The following are Likert questions about this course:

- I will do well in this course
- I am excited about this course
- I think learning to program can help me with other classes
- I think learning to program will help me with things outside of school
- I think about the programs that control the devices I use in my everyday life

The following are short answer questions about this course:

- Why are you taking this course?
- What do you hope to learn in this course?
- The thing I am most excited about for this class is:
- Do you have any friends taking this course? If so, how many?
- How did you hear about this course?
- To be successful in programming courses, students need to:
- I define programming as:
- The most important thing about programming is:
- The hardest thing about programming is:

The following are questions about questions about students' prior experience with programming and technology:

- How much time do you spend on a computer at home each day?
    - I don't use a computer
    - Less then 1 hour
    - Between 1 and 2 hours
    - Between 2 and 3 hours
    - More then 3 hours
- What do you do on the computer outside of school?
- What types of computational devices do you own/use regularly?
    - Laptop computer
    - Desktop computer
    - Tablet (iPad, Surface, etc.)
    - Smartphone (iPhone, Samsun Galaxy, etc)
    - Portable Media Player (iPod, portable movie player, etc.)
    - Game console (Xbox, Play Station, Wii, etc.)
- Have you taken any programming courses previously? If so, what coure(s) and when?
- Have you ever used these languages/programming tools?
    - Scratch or Snap!
    - App Inventor
    - Alice
    - HTML, CSS or JavaScript
    - Java, C++ or C#
    - Python, Lisp or Scheme
    - Other:
- Do you know any professional programmers? If so, who?

Mid Survey

The following questions are 1-10 point Likert scale questions about programming and computer science generally

- Programming is fun
- I am good at programming
- Programming is hard
- I know more than my friends about programming
- Most women can learn to program
- Programming jobs are boring
- I like programming
- Most men can learn to program
- My family encourages me to learn to program
- My friends like using computers
- I can become good at programming
- I like the challenge of programming
- I think programming is useful
- I am interested in a career in programming
- Knowing how to program is important
- I think learning to program can help me with other classes
- I think learning to program will help me with things outside of school
- I cannot learn to program well if the teacher does not explain things well
- I think about the programs that control the devices I use in my everyday life.
- I plan to take more computer science courses after this one.

The following questions are 1-10 point Likert scale questions about this course:

- I learned a lot about programming using Snappier!
- After using Snappier! for 5 weeks, I am a better programmer
- I am more confident as a programmer because of Snappier!
- Snappier! has made me ready to learn to program in Java
- I will do well in this course
- I am excited about what is coming next in this course
- I am more excited about programming now than I was at the start of the year.

The following are short answer questions about this course:

- How is this course different than what you expected at the start of the year?
- What have you learned so far in this course?
- The thing I have enjoyed the most in the course so far is:
- The thing I have enjoyed the least in the course so far is:
- The thing I am most looking forward during the rest of the year in this course is:
- The thing I learned in snap! that will be most useful in Java is:

- The thing will be the most different about programming in Java compared to programming in snap! is:
- To be successful in programming courses, students need to:
- I define programming as:
- The most important thing about programming is:
- The hardest thing about programming is:

Post Survey

Appendix 2 – CS Content Assessment

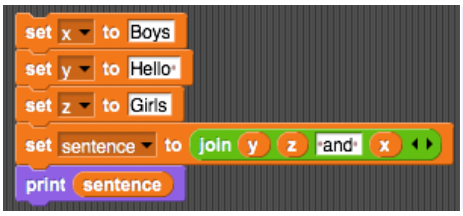
Variables

What are the values of x and y after this script runs?


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

a - x = 15, y = 15
b – x = 5, y = 10
c – x = 15, y = 10
d – x = 15, y = “x”

What will be printed after running this script?




```
var x, y, z, sentence;
x = "Boys";
y = "Hello ";
z = "Girls";
sentence = y + z + " and " + x;
print(sentence);
```

a – “join y z and x”
b – “sentence”
c - “BoysHello Girls”
d – “Hello Boys and Girls”
e – “Hello Girls and Boys”

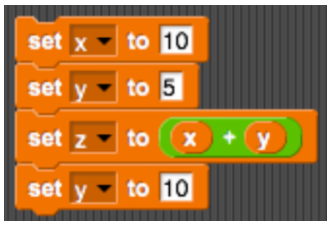
What are x, y and z equal to after this program is run?

```
var x, y, z;
x = "yes";
y = x;
z = x;
x = "no";
y = "maybe";
z = x;
```

a) x = “yes”, y = “maybe”, z = “no”
b) x = “no”, y = “maybe”, z = “yes”
c) x = “no”, y = “maybe”, z = “x”
d) x = “yes”, y=”maybe”, z= “yes”
e) x = “no”, y = “maybe”, z = “no”

What are x, y and z equal to after this program is run?

```
var x = 10;
var y = 5;
var z = x + y;
y = 10;
```



a) x = 10, y = 5, z = 15
b) x = 10, y = 10, z = 20
c) x = 10, y = 10, z = 15
d) x = 10, y = 10, z = "x + y"

Conditional logic

What is the behavior of this script?


```
if (x > 10) {
  x = 10;
} else {
  if (x < 5) {
    x = 5;
  }
}
```

a – Makes sure the value of x is not equal to 10
b – Makes sure the value of x is less than 5
c – Makes sure the value of x is between 10 and 5
d – It always sets x equal to 5

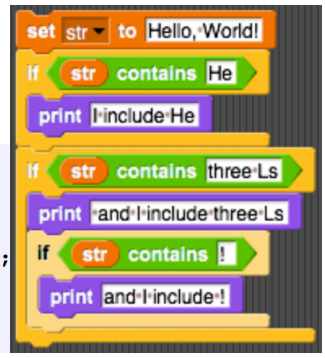
What will be printed after running this script?


```
var x = 3;
if (x > 7) {
  x = 7;
} else {
  if (x < 1) {
    x = 1;
  }
}
print("x is " + x);
```

A – "x is 3"
b – "5"
c – "x is 5"
d – "x is x"

What will be printed after running this program?

```
var str = "Hello, World!";
if (str.contains("He")) {
  print("I include He");
}
if (str.contains("three Ls"))
  print(" and I include three Ls");
}
if (str.contains("!")) {
  print(" and I include !");
}
```



a – “I include He”
b – “ and I include !”
c – “I include He and I include three Ls and I include !”
d – “I include He and I include !”
e – “” [nothing will be printed]

Given the following program where `nums_list` is a list of numbers:

```
if (nums_list.length < 2) {
  print("the list is too short!");
} else {
  if (nums_list.length > 6) {
    print ("the list is too long!");
  } else if (nums_list.length > 4) {
    print ("the list is just right");
  } else {
    print ("the list is still too short!");
  }
}
```

If `nums_list` was a list including the following numbers: 1, 2, 3, 4, and 5, what would be printed?

a – “the list is too short!”
b – “the list it too long!”
C – “the list is just right”
D – “the list is still too short!”
E – “the list is just rightthe list is still too short!”

If `nums_list` was changed to include only the number 1, what would be printed?
a – “the list is too short!”
b – “the list it too long!”
C – “the list is just right”
D – “the list is still too short!”
E – “the list is too short!the list is still too short!”

Iteration/Loops

How many times will the word “here” be printed when this script is run?


```
for (var i = 0; i < 2; i++) {
  print("here");
  for (var j = 0; j < 3; j++) {
    print("here");
  }
  print("here");
}
```

a – 3
b – 6
c – 10
d – 18
e – it will be different each time you run it

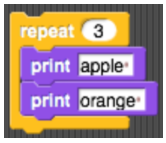
How many times will the word “here” be printed when this script is run?


```
var x = 0;
while (x < 10) {
  print("here");
}
x = (x + 2);
```

a – 0
b – 5
c – 10
d – “here” will be continuously printed until the script is stopped

What will be printed when this program is run?

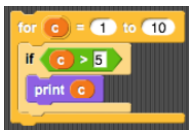
```
for (var c = 0; c < 3; c++) {
  print("apple ");
  print("orange ");
}
```



a) “apple apple apple orange orange orange ”
b) “apple orange ”
c) “apple orange apple orange apple orange ”
d) “” [nothing with be printed]

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

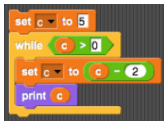
```
for (var c = 0; c < 10; c++) {
  if (c > 5) {
    print (c);
  }
}
```



a – 1
b – 5
c – 10
d – it will be different each time you run it

What will be printed when this program is run?

```
var c = 5;
while (c > 0) {
  c = c - 2;
  print(c);
}
```



a - “543210”
b – “531”
c – “31-1”
d – “31”

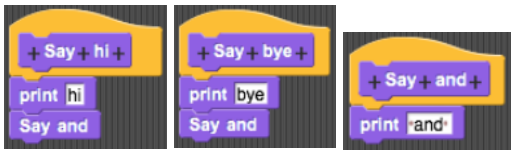
Functions

Here are three functions that each print something different:

```
function say_hi() {
  print("hi");
  say_and();
}

function say_bye() {
  print("bye");
  say_and();
}

function say_and() {
  print(" and ");
}
```



What is printed when this program is run?

```
say_hi();
say_and();
say_bye();
```

a) “hi and bye”
b) “hi and and bye and ”
c) “hi and bye and”
d) “hi bye and”

Here are three functions that each perform a mathematical operation:

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

function op2(c) {
  return op1(c, c);
}

function op3(d) {
  return d * d;
}
```

What is output of this program?

```
print(op2(10));
```

a) “10”
b) “1010”
c) “20”
d) [nothing is printed]

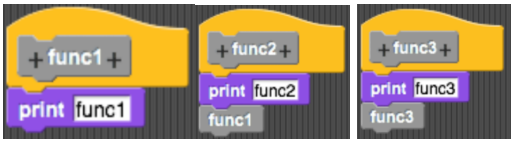
e) this program would cause an error

What is the output of this program?

```
print(op1(op3(4),5));
```

a) “21”
b) “9”
c) “81”
d) [nothing is printed]
e) this program would cause an error

Here are three events/functions:




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

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

```
function func3() {
  print("func3");
  func3();
}
```

What is printed when this script is run?


```
func1();
func2();
```

a) “op1 op2”
b) “op1 op2 op1”
c) “broadcast op1 broadcast op2”
d) “” [nothing is printed]

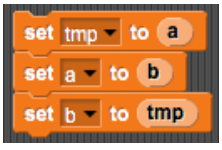
What is printed when this script is run?


```
func3();
func1();
```

a) “op3 op3 op1”
b) “op3 op1 op3”
c) “op3 ” will be printed over and over until the script is stopped
d) “” (nothing is printed, this script will not work)

Code Comprehension

`a`, `b`, and `tmp` are variables. What does this script do?

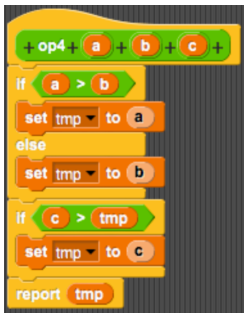



```
tmp = a;
a = b;
b = tmp;
```

a) makes a and b equal to each other
b) swaps the values of a and b
c) this script doesn't do anything
d) rearranges the variables a, b, and tmp.

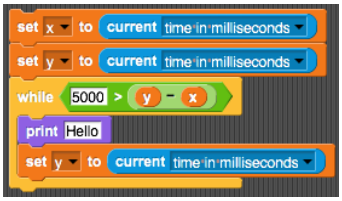
The function op4 takes in 3 numbers, what does this function do?

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



a) Randomly returns on of the three numbers
b) Returns the largest of the three numbers
c) Set all three inputs to a temporary value
d) Return the smallest of the three numbers

What does this script do?

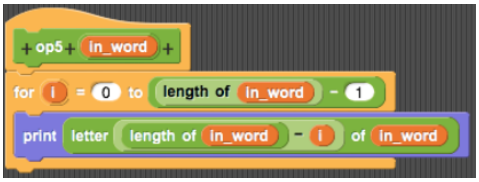



```
var x = new Date().getMilliseconds();
var y = new Date().getMilliseconds();
while (5000 > (y - x)) {
  print("Hello");
  y = new Date().getMilliseconds();
}
```

a) Prints "Hello" once
b) Prints "Hello" 5000 times
c) Prints "Hello" continuously for 5 seconds
d) Prints "Hello" once after 5 seconds has elapsed

This function takes a word as an input, what does this function do with that word?

```
function op5(in_word) {
  for(var c = 0; c <= in_word.length; c++) {
    print(in_word.characterAt(in_word.length - c));
  }
}
```



a) Prints the word once for each letter of the word (so for a 3-letter word, the whole word is printed 3 times)
b) Prints the word one letter at a time in reverse order
c) Prints the word one letter at a time in the original order

d) Prints the last letter of the word that is passed in

Algorithms

If you want to write a program that asks a user to type in a sentence, then reports back to the user the number of times the letter 'e' appears in that sentence, which of these things would your programming language **not** need to be able to do:

1. Compare two letters to each other to determine if they are the same
2. Display text on the screen
3. Create and modify data as a program runs
4. Convert letters into numbers and numbers into letters
5. Store user entered information

You are writing a guessing game program. Your program randomly choses a number between 1 and 100, then asks the player to guess the number. If the player's guess is too high the program will tell them to guess lower, if their guess is too low, the program will tell them to guess higher. The player gets 10 guesses. Below are the six necessary parts of the program, but in a jumbled order.

A – Compare the guess to the mystery number and report back 'Too High', 'Too Low!', or 'You Guessed it!'
B – Increase the guess counter by one to record the guess attempt
C – Check to see if the player has used up all 10 guesses, if so, print: 'You are out of Guesses!"
D – Ask the user for their guess and then read it in
E – Ask the user if they would like to play again
F – Randomly choose the mystery number and set the user's guess_counter to 0

Please answer the following true/false questions:

T/F - Part F must be the first step in the program
T/F - It is necessary that part A comes before part B
T/F - Part C must come before part E
T/F - Part B is optional; the game will work without it
T/F - There is only one way to write the code for these six parts of the program so that the game works

Because the guessing game gives the player 10 tries, some of the steps in the program will need to be run once for each guess the player makes, while other steps will only be run once per game. Which steps will only be run once for each time the game is played?

1. Only E
2. B, C, and E

3. E and F
4. A, B, C, and D

Imagine you are writing a computer program that will ask you for a class period, then print what class you have that period on the screen. When you try and use the program, you realize it always prints out your first period class. Which of the following potential 'bugs' **could not** cause this error:

1. You are reading in the class period incorrectly
2. The steps you follow to match the input period to your stored schedule is incorrect
3. You are using the wrong command to print words onto the screen
4. You accidentally stored your first period class for every period of the day

Below is a program that will read in the names of all students in a class, and then print out a list of all the students whose first name ends with the letter 'a':

Here are five possible steps that could be used to finish the program

A. Check to see if the student's first name ends in 'a'
B. Print out the students names
C. Add the students name to a_names
D. Remove the student's name from all_students
E. End the program

Which steps should replace <do-something-here> so the program will work correctly:

1. A then B then C
2. C then D then E
3. A then C
4. Just C

When the program is working correctly, which statement will always be true?

1. No two students will have the same name
2. a_names has fewer or the same number of names in it as all_students
3. There will be no student names in a_names
4. If you run the program twice for the same class, you will get a different list of names printed out each time

Appendix 3 – Student Interview Protocols

These interviews will be semi-structured clinical interviews. As such, the questions below are intended (for the most part) to provoke longer discussions of a back-and-forth nature with the interviewer pursuing avenues of inquiry opened up by student responses.

Initial Interview (weeks 1 & 2)

Background questions

- Tell me about your programming experience?
- For students who have taken Exploring Computer Science (ECS) students:
    - Tell me about ECS
    - What did you like about the class?
    - What didn't you like?
    - What do you think the goal was?
    - What did you learn?
    - How did it prepare you for this course?
    - Tell me about the projects in the class
- Tell me about your Scratch experience?
- Do you have any snap experience?

Impressions of blocks-based programming

- Is this (snap!) programming – what about it makes it programming? What about it makes it different from 'real programming'
- Go through blocks – explain what he thinks they do
- Have them write a program to set a variable to a random number then print it out
- Show them a simple blocks program, then a textual equivalent
    - What are the strengths and weakness of each way of programming?
    - Is one easier than the other? Why?
- We start this course by having you work with these blocks-based environments before moving to text programming, why do you think we do that?

Concluding Introductory Interview (week 5)

Overview

- How has the class been going so far?
- What have you liked? And what have you not liked?
- If a friend of yours asked you to describe what you have done in class so far this year, what would you tell them?

Transition to java

- How is what you have done so far in Snappier! different than what you think you'll be doing for the rest of the year?

- What do you think you have done so far this year that you think will be helpful for starting to program in Java next week?

Concepts

- Can you tell me what a variable is?
    - When would you use a variable?
- Can you tell me what a predicate is?
    - When would you use a predicate?
- Can you tell me what a conditional statement?
    - When would you use a conditional statement?
- Can you tell me what a repeat block does?
    - When would you use a repeat block?
- Can you tell me what a function is?
    - When would you use a function?

Reading/Writing Programs

- Read/explain two program programs one in JavaScript and one in blocks
    - One program repeats back a word one letter at a time unless it's too long
    - One program starts with a random number then counts down to 0 by a different amount depending on how big the initial number is

- Have them compose a blocks program for me
    - Pick a note level between 50 and 70 and play every other note until you hit 80. At the end, say if you played more or less than 10 notes
- Concluding question: is this program similar to either of the two we looked at before?

Concluding Interview (weeks ~10 and ~20)

Overview

- How is the class been going so far?
- Do you like Java?
- What have you liked? And what have you not liked?
- If a friend of yours asked you to describe what you have done in class so far this year, what would you tell them?

Comparison/Transition to java

- What do you think are the big differences between Java and Snappier?
- How is what you have done so far in Java different than what you did in the first part of the year in Snappier?
    - What is the same between Java and Snappier?
- Do you think the stuff you did in snappier! was helpful for what you're doing now?
    - If so, what and how has it helped?
    - Is there anything from Snap that you think made Java harder?
- Are there any strategies for programming that you developed while using snap that you now use in Java?

    - Any strategies that are different in Java?
- You have worked in both java and in blocks
    - Which format do you find easier to read programs in? Why?
    - Which format do you find easier to write programs in? Why?

Concept Map

- Can you map out for me what you think and the central ideas of programming and how they relate to each other?

Java Questions

- What have you learned so far about programming in Java?
- What types of things can you do with Java?
    - What types of things could you do with snappier?
- What is it important to know to be good at Java?
- What do you think the easiest thing
- Do you like programming in Java?
    - Did you prefer programming with snappier!?

Java Concepts

[note: we will only ask questions about concepts that have been covered]

- Can you tell me what a variable is?
    - When would you use a variable?
    - Are there any differences between variables in snappier and Java?
- Can you tell me what an object is?
    - What do you do with an object?
    - Did we have objects in snap?
- Can you tell me what a method is?
    - What do you do with a method?
    - Did we have methods in snap?
- Can you tell me what a conditional statement?
    - When would you use a conditional statement?
    - Did we have methods in snap?
- Can you tell me what a for loop does?
    - When would you use a for loop?
    - Did we have methods in snap?
- Can you tell me what a function is?
    - When would you use a function?
    - Did we have methods in snap?

Do you remember and blocks or ideas from snap! that you haven't used yet in Java?

Bibliography

Abelson, H., & DiSessa, A. A. (1986). *Turtle geometry: The computer as a medium for exploring mathematics*. The MIT Press.

ACM/IEEE-CS Joint Task Force on Computing Curricula. (2013). *Computer Science Curricula 2013*. ACM Press and IEEE Computer Society Press.

Adams, J. (2007). *Alice in Action with Java* (1 edition.). Boston, Mass: Cengage Learning.

Allen, E., Cartwright, R., & Stoler, B. (2002). DrJava: A lightweight pedagogic environment for Java. *ACM SIGCSE Bulletin*, *34*(1), 137–141.

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

App Inventory Java Bridge. (2014). Retrieved from https://code.google.com/p/apptomarket/

Armoni, M., & Ben-Ari, M. (2010). *Computer Science Concepts in Scratch*. Retrieved from http://onlinelibrary.wiley.com/doi/10.1002/app.1975.070190908/abstract

Baker, R. S., & Yacef, K. (2009). The state of educational data mining in 2009: A review and future visions. *JEDM-Journal of Educational Data Mining*, *1*(1), 3–17.

Bakhtin, M. M. (1981). *The dialogic imagination: four essays*. Austin: University of Texas Press.

Baroth, E. C., & Hartsough, C. (1995). Experience Report: Visual Programming in the Real World. *Visual Object Oriented Programming, Edited by MM Burnett, A. Goldberg & TG Lewis, Manning Publications, Prentice Hall*, 21–42.

Begel, A. (1996). *LogoBlocks: A graphical programming language for interacting with the world.* Electrical Engineering and Computer Science Department. MIT, Cambridge, MA.

Begel, A., & Klopfer, E. (2007). Starlogo TNG: An introduction to game development. *Journal of E-Learning*.

Behnke, K. A. (2013). SLASH: Scratch-based visual programming in Second Life for introductory computer science education Poster Session. In *Proceeding of the 44th ACM technical symposium on Computer science education*. Denver, CO.

Berland, M., Martin, T., Benton, T., Petrick Smith, C., & Davis, D. (2013). Using Learning Analytics to Understand the Learning Pathways of Novice Programmers. *Journal of the Learning Sciences*, *22*(4), 564–599. doi:10.1080/10508406.2013.836655

Blackwell, A. F., Whitley, K. N., Good, J., & Petre, M. (2001). Cognitive factors in programming with diagrams. *Artificial Intelligence Review*, *15*(1-2), 95–114.

Blikstein, P., & Wilensky, U. (2009). An Atom is Known by the Company it Keeps: A Constructionist Learning Environment for Materials Science Using Agent-Based Modeling. *International Journal of Computers for Mathematical Learning*, *14*(2), 81–119.

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. *Journal of the Learning Sciences*, *0*(ja), null. doi:10.1080/10508406.2014.954750

Bonar, J., & Liffick, B. W. (1987). A visual programming language for novices. In S. K. Chang (Ed.), *Principles of Visual Programming Systems*. Prentice-Hall, Inc.

Bontá, P., Papert, A., & Silverman, B. (2010). Turtle, Art, TurtleArt. In *Proceedings of Constructionism 2010 Conference*. Paris, France.

Boroditsky, L. (2001). Does Language Shape Thought?: Mandarin and English Speakers' Conceptions of Time. *Cognitive Psychology*, *43*(1), 1–22. doi:10.1006/cogp.2001.0748

Bruckman, A. (1997). *MOOSE Crossing: Construction, community, and learning in a networked virtual world for kids*. MIT.

Bruckman, A., Biggers, M., Ericson, B., McKlin, T., Dimond, J., DiSalvo, B., … Yardi, S. (2009). Georgia computes!: Improving the computing education pipeline. In *ACM SIGCSE Bulletin* (Vol. 41, pp. 86–90). ACM.

Bruckman, A., & Edwards, E. (1999). Should we leverage natural-language knowledge? An analysis of user errors in a natural-language-style programming language. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems* (pp. 207–214). ACM.

Bruckman, A., Jensen, C., & DeBonte, A. (2002). Gender and Programming Achievement in a CSCL Environment. In *Proceedings of the Conference on Computer Support for Collaborative Learning: Foundations for a CSCL Community* (pp. 119–127). Boulder, Colorado: International Society of the Learning Sciences.

Brusilovsky, P., Calabrese, E., Hvorecky, J., Kouchnirenko, A., & Miller, P. (1997). Mini-languages: a way to learn programming principles. *Education and Information Technologies*, *2*(1), 65–83.

Buechley, L., & Eisenberg, M. (2008). The LilyPad Arduino: Toward wearable engineering for everyone. *Pervasive Computing, IEEE*, *7*(2), 12–15.

Buechley, L., Eisenberg, M., Catchen, J., & Crockett, A. (2008). The LilyPad Arduino: using computational textiles to investigate engagement, aesthetics, and diversity in computer science education. In *Proceedings of the SIGCHI conference on Human factors in computing systems* (pp. 423–432). ACM.

Burke, Q., & Kafai, Y. B. (2010). Programming & storytelling: opportunities for learning about coding & composition. In *Proceedings of the 9th International Conference on Interaction Design and Children* (pp. 348–351). ACM.

Chadha, K. (2014). *Improving App Inventor through Conversion between Blocks and Text* (Honors Thesis). Wellesley College.

Chandhok, R. P., & Miller, P. L. (1989). The design and implementation of the Pascal Genie. In *Proceedings of the 17th conference on ACM Annual Computer Science Conference* (pp. 374–379). ACM.

Chetty, J., & Barlow-Jones, G. (2012). Bridging the Gap: the Role of Mediated Transfer for Computer Programming. *International Proceedings of Computer Science & Information Technology*, *43*.

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

Collective, T. D.-B. R. (2003). Design-based research: An emerging paradigm for educational inquiry. *Educational Researcher*, 5–8.

Collins, A., Joseph, D., & Bielaczyc, K. (2004). Design research: Theoretical and methodological issues. *The Journal of the Learning Sciences*, *13*(1), 15–42.

Colmerauer, A. (1985). Prolog in 10 Figures. *Commun. ACM*, *28*(12), 1296–1310. doi:10.1145/214956.214958

Confrey, J., & Smith, E. (1994). Exponential functions, rates of change, and the multiplicative unit. *Educational Studies in Mathematics*, *26*(2-3), 135–164. doi:10.1007/BF01273661

Cooper, S., & Cunningham, S. (2010). Teaching computer science in context. *ACM Inroads*, *1*(1), 5–8.

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

Cooper, S., Dann, W., & Pausch, R. (2003). Teaching objects-first in introductory computer science. In *Proceedings of the 34th SIGCSE technical symposium on Computer science education* (p. 195).

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

Dann, W., Cooper, S., & Ericson, B. (2009). *Exploring Wonderland: Java Programming Using Alice and Media Computation*. Prentice Hall Press.

Dann, W., Cooper, S., & Pausch, R. (2011). *Learning to Program with Alice*. Prentice Hall Press.

Dann, W., Cosgrove, D., Slater, D., Culyba, D., & Cooper, S. (2012). Mediated transfer: Alice 3 to Java. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education* (pp. 141–146). ACM.

Denny, P., Luxton-Reilly, A., Tempero, E., & Hendrickx, J. (2011). Understanding the syntax barrier for novices. In *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education* (pp. 208–212). ACM.

Dijkstra, E. W. (1982). How do we tell truths that might hurt? In *Selected Writings on Computing: A Personal Perspective* (pp. 129–131). Springer.

DiSalvo, B., Guzdial, M., Bruckman, A., & McKlin, T. (2014). Saving Face While Geeking Out: Video Game Testing as a Justification for Learning Computer Science. *Journal of the Learning Sciences*, *23*(3), 272–315. doi:10.1080/10508406.2014.893434

diSessa, A. A. (2000). *Changing minds: Computers, learning, and literacy*. Cambridge, MA: MIT Press.

diSessa, A. A., & Abelson, H. (1986). Boxer: a reconstructible computational medium. *Communications of the ACM*, *29*(9), 859–868.

diSessa, A. A., Hammer, D., Sherin, B. L., & Kolpakowski, T. (1991). Inventing graphing: Meta-representational expertise in children. *Journal of Mathematical Behavior*, *10*, 117–160.

Donzeau-Gouge, V., Huet, G., Lang, B., & Kahn, G. (1984). Programming environments based on structured editors: The MENTOR experience. In D. Barstow, H. E. Shrobe, & E. Sandewall (Eds.), *Interactive Program- ming Environments*. McGraw Hill.

Duncan, C., Bell, T., & Tanimoto, S. (2014). Should Your 8-year-old Learn Coding? In *Proceedings of the 9th Workshop in Primary and Secondary Computing Education* (pp. 60–69). New York, NY, USA: ACM. doi:10.1145/2670757.2670774

Feurzeig, W., Papert, S., Bloom, M., Grant, R., & Solomon, C. (1969). *Programming-languages as a conceptual framework for teaching mathematics* (BBN Report No. 1889). Cambridge, MA: Bolt, Beranek, and Newman.

Feurzeig, W., Papert, S., & Lawler, B. (2011). Programming-languages as a conceptual framework for teaching mathematics. *Interactive Learning Environments*, *19*(5), 487–501.

Findler, R. B., Clements, J., Flanagan, C., Flatt, M., Krishnamurthi, S., Steckler, P., & Felleisen, M. (2002). DrScheme: A programming environment for Scheme. *Journal of Functional Programming*, *12*(02), 159–182.

Fisher, A., Margolis, J., & Miller, F. (1997). Undergraduate women in computer science: experience, motivation and culture. In *ACM SIGCSE Bulletin* (Vol. 29, pp. 106–110). ACM.

Fraser, N. (2013). *Blockly*. https://code.google.com/p/blockly: Google.

Gall, H. C., Fluri, B., & Pinzger, M. (2009). Change analysis with evolizer and changedistiller. *IEEE Software*, *26*(1), 26–33.

Garlan, D. B., & Miller, P. L. (1984). GNOME: An introductory programming environment based on a family of structure editors. *ACM Sigplan Notices*, *19*(5), 65–72.

Garlick, R., & Cankaya, E. C. (2010). Using Alice in CS1: A quantitative experiment. In *Proceedings of the fifteenth annual conference on Innovation and technology in computer science education* (pp. 165–168). ACM.

Goodman, D. (1988). *The Complete HyperCard Handbook* (2nd ed.). New York: Bantam Books.

Green, T. R. G. (1977). Conditional program statements and their comprehensibility to professional programmers. *Journal of Occupational Psychology*, *50*(2), 93–109.

Green, T. R. G., & Petre, M. (1992). When visual programs are harder to read than textual programs. In *Human-Computer Interaction: Tasks and Organisation, Proceedings of ECCE-6 (6th European Conference on Cognitive Ergonomics). GC van der Veer, MJ Tauber, S. Bagnarola and M. Antavolits. Rome, CUD*. Citeseer.

Green, T. R. G., & Petre, M. (1996). Usability analysis of visual programming environments: A "cognitive dimensions" framework. *Journal of Visual Languages and Computing*, *7*(2), 131–174.

Green, T. R. G., Petre, M., & Bellamy, R. K. E. (1991). Comprehensibility of visual and textual programs: A test of superlativism against the'match-mismatch'conjecture. *ESP*, *91*(743), 121–146.

Guzdial, M. (2004). Programming environments for novices. *Computer Science Education Research*, *2004*, 127–154.

Guzdial, M. (2010). Does contextualized computing education help? *ACM Inroads*, *1*(4), 4–6.

Hancock, C. M. (2003). *Real-time programming and the big ideas of computational literacy*. Citeseer.

Harvey, B., & Mönig, J. (2010). Bringing "no ceiling" to Scratch: Can one language serve kids and computer scientists? In J. Clayson & I. Kalas (Eds.), *Proceedings of Constructionism 2010 Conference* (pp. 1–10). Paris, France.

Henriksen, P., & Kölling, M. (2004). Greenfoot: Combining object visualisation with interaction. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications* (pp. 73–82).

Hils, D. D. (1992). Visual languages and computing survey: Data flow visual programming languages. *Journal of Visual Languages & Computing*, *3*(1), 69–101.

Holt, R. C., & Cordy, J. R. (1988). The Turing Programming Language. *Commun. ACM*, *31*(12), 1410–1423. doi:10.1145/53580.53581

Hopscotch. (2014). New York, NY: Hopscotch. Retrieved from http://www.gethopscotch.com/

Horn, M. S., Solovey, E. T., Crouser, R. J., & Jacob, R. J. . (2009). Comparing the use of tangible and graphical programming languages for informal science education. In *Proceedings of the 27th international conference on Human factors in computing systems* (pp. 975–984). ACM Press.

Horstmann, C. S. (2012). *Java Concepts: Early Objects* (7 edition.). Hoboken, NJ: Wiley.

Hour of Code. (2013). Code.org. Retrieved from http://code.org/learn

Howland, K., & Good, J. (2014). Learning to communicate computationally with flip: A bi-modal programming language for game creation. *Computers & Education*. doi:10.1016/j.compedu.2014.08.014

Hoyles, C., & Noss, R. (2004). Situated abstraction: mathematical understandings at the boundary. *Proceedings of Study Group 22 of ICME-10*, *7*, 212–224.

Hundhausen, C. D., Farley, S. F., & Brown, J. L. (2009). Can direct manipulation lower the barriers to computer programming and promote transfer of training? *ACM Transactions on Computer-Human Interaction*, *16*(3), 1–40. doi:10.1145/1592440.1592442

Hutchins, E. L., Hollan, J. D., & Norman, D. A. (1985). Direct manipulation interfaces. *Human-Computer Interaction*, *1*(4), 311–338.

Ioannidou, A., Repenning, A., & Webb, D. C. (2009). AgentCubes: Incremental 3D end-user development. *Journal of Visual Languages & Computing*, *20*(4), 236–251.

Jadud, M. C. (2005). A first look at novice compilation behaviour using BlueJ. *Computer Science Education*, *15*(1), 25–40.

Jadud, M. C., & Henriksen, P. (2009). Flexible, reusable tools for studying novice programmers. In *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. In *IEEE 21st Conference on Software Engineering Education and Training, 2008. CSEET '08* (pp. 129–136). doi:10.1109/CSEET.2008.35

Johnson, G. W. (1997). *LabVIEW graphical programming: practical applications in instrumentation and control*. McGraw-Hill School Education Group.

Kahn, K. (1999). From prolog to Zelda to ToonTalk. In *Proceedings of the International Conference on Logic Programming* (pp. 67–78).

Kay, A. (2005). Squeak etoys, children & learning. *Online Article*, *2006*. Retrieved from ftp://ofset2.unix-ag.uni-kl.de/speeches/jrfernandez/malaga08/doc/etoys_n_learning.pdf

Kelleher, C., & Pausch, R. (2005). Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys*, *37*(2), 83–137.

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

Kemeny, J. G., & Kurtz, T. E. (1980). *Basic programming*. John Wiley & Sons, Inc.

Kölling, M., Quig, B., Patterson, A., & Rosenberg, J. (2003). The BlueJ System and its Pedagogy. *Computer Science Education*, *13*(4), 249–268.

Kölling, M., & Rosenberg, J. (1996). Blue—a language for teaching object-oriented programming. In *ACM SIGCSE Bulletin* (Vol. 28, pp. 190–194). ACM.

Lave, J. (1988). *Cognition in practice: Mind, mathematics, and culture in everyday life*. Cambridge Univ Pr.

Lego Systems Inc. (2008). *Lego Mindstorms NXT-G Invention System*. Retrieved from http://mindstorms.lego.com

Levy, S. T., & Wilensky, U. (2011). Mining students' inquiry actions for understanding of complex systems. *Computers & Education*, *56*(3), 556–573.

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

Lewis, J., & DePasquale, P. (2008). *Programming with Alice and Java*. Boston: Addison-Wesley.

Luria, A. R. (1982). *Language and cognition*. (J. V. Wertsch, Ed.). Winston ; Wiley, Washington, D.C. : New York ; Chichester :

Made with Code. (2014). Google. Retrieved from https://www.madewithcode.com/

Malan, D. J., & Leitner, H. H. (2007). Scratch for budding computer scientists. In *ACM SIGCSE Bulletin* (Vol. 39, pp. 223–227). ACM.

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

Margolis, J. (2008). *Stuck in the shallow end: Education, race, and computing*. The MIT Press.

Margolis, J., & Fisher, A. (2003). *Unlocking the clubhouse: Women in computing*. The MIT Press.

Meerbaum-Salant, O., Armoni, M., & Ben-Ari, M. (2011). Habits of programming in Scratch. In *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education* (pp. 168–172). Darmstadt, Germany: ACM.

Meerbaum-Salant, O., Armoni, M., & Ben-Ari, M. M. (2010). Learning computer science concepts with scratch. In *Proceedings of the Sixth international workshop on Computing education research* (pp. 69–76).

Mendelsohn, P., Green, T. R. G., & Brna, P. (1990). *Programming languages in education: The search for an easy start*. Academic Press London.

Miller, P., Pane, J., Meter, G., & Vorthmann, S. (1994). Evolution of novice programming environments: the structure editors of Carnegie Mellon University. *Interactive Learning Environments*, *4*(2), 140–158.

Modrow, E., Mönig, J., & Strecker, K. (2011). Wozu JAVA? *LOG IN*, *31*(168), 35–41.

Moher, T. G., Mak, D. C., Blumenthal, B., & Levanthal, L. M. (1993). Comparing the comprehensibility of textual and graphical programs. In *Empirical Studies of Programmers: Fifth Workshop* (pp. 137–161). Ablex, Norwood, NJ.

Moskal, B., Lurie, D., & Cooper, S. (2004). Evaluating the effectiveness of a new instructional approach. In *Proceedings of the 35th SIGCSE technical symposium on Computer science education* (pp. 75–79).

Motil, J., & Epstein, D. (1998). JJ: a Language Designed for Beginners. Retrieved from http://www.csun.edu/~jmotil/BeginLanguageJr.pdf

Mullins, P., Whitfield, D., & Conlon, M. (2009). Using Alice 2.0 as a first language. *Journal of Computing Sciences in Colleges*, *24*(3), 136–143.

Myers, B. A. (1990). Taxonomies of visual programming and program visualization. *Journal of Visual Languages & Computing*, *1*(1), 97–123.

Nemirovsky, R. (1994). On ways of symbolizing: The case of Laura and the velocity sign. *The Journal of Mathematical Behavior*, *13*(4), 389–422.

Norman, D. A. (1993). *Things that make us smart: Defending human attributes in the age of the machine*. Basic Books.

Noss, R., Healy, L., & Hoyles, C. (1997). The construction of mathematical meanings: Connecting the visual with the symbolic. *Educational Studies in Mathematics*, *33*(2), 203–233.

Noss, R., & Hoyles, C. (1996). *Windows on mathematical meanings: Learning cultures and computers*. Dordrecht: Kluwer.

Ong, W. (1982). *Orality and Literacy: The technologizing of the world*. London: Routledge.

Palmer, S. E. (1978). Fundamental aspects of cognitive representation. In E. Rosch & B. B. Lloyd (Eds.), *Cognition and categorization* (Vol. 259, pp. 259–303). Hillsdale, N.J.: Lawrence Erlbaum Associates.

Pane, J., & Miller, P. (1993). The ACSE multimedia science learning environment. In *Proceedings of the 1993 International Conference on Computers in Education* (pp. 168–173).

Papadimitriou, C. H. (2003). MythematiCS: in praise of storytelling in the teaching of computer science and math. *SIGCSE BULLETIN*, *35*(4), 7–9.

Papert, S. (1980). *Mindstorms: Children, computers, and powerful ideas*. New York: Basic books.

Parsons, D., & Haden, P. (2007). Programming osmosis: Knowledge transfer from imperative to visual programming environments. In S. Mann & N. Bridgeman (Eds.), *Procedings of The Twentieth Annual NACCQ Conference* (pp. 209–215). Hamilton, New Zealand.

Pattis, R. E. (1981). *Karel the robot: a gentle introduction to the art of programming*. John Wiley & Sons, Inc.

Pears, A., Seidman, S., Malmi, L., Mannila, L., Adams, E., Bennedsen, J., … Paterson, J. (2007). A survey of literature on the teaching of introductory programming. In *ACM SIGCSE Bulletin* (Vol. 39, pp. 204–223). ACM.

Petre, M. (1995). Why looking isn't always seeing: readership skills and graphical programming. *Communications of the ACM*, *38*(6), 33–44.

PicoBlocks. (2008). Playful Invention Company. Retrieved from http://www.picocricket.com/download.html

Powers, K., Ecott, S., & Hirshfield, L. M. (2007). Through the looking glass: teaching CS0 with Alice. *ACM SIGCSE Bulletin*, *39*(1), 213–217.

Repenning, A., Ioannidou, A., & Zola, J. (2000). AgentSheets: End-user programmable simulations. *Journal of Artificial Societies and Social Simulation*, *3*(3).

Repenning, A., & Sumner, T. (1995). Agentsheets: A medium for creating domain-oriented visual languages. *IEEE Computer*, *28*(3), 17–25.

Resnick, M., Silverman, B., Kafai, Y., Maloney, J., Monroy-Hernández, A., Rusk, N., … Silver, J. (2009). Scratch: Programming for all. *Communications of the ACM*, *52*(11), 60.

Resnick, M., & Wilensky, U. (1993). Beyond the deterministic, centralized mindsets: New thinking for new sciences. In *annual meeting of the American Educational Research Association, Atlanta, GA*.

Robins, A., Rountree, J., & Rountree, N. (2003). Learning and Teaching Programming: A Review and Discussion. *Computer Science Education*, *13*(2), 137. doi:10.1076/csed.13.2.137.14200

Roque, R. V. (2007). *OpenBlocks: An extendable framework for graphical block programming systems* (Master's Thesis). Massachusetts Institute of Technology.

Salton, G., Wong, A., & Yang, C. S. (1975). A Vector Space Model for Automatic Indexing. *Commun. ACM*, *18*(11), 613–620. doi:10.1145/361219.361220

Sammet, J. E. (1981). The Early History of COBOL. In R. L. Wexelblat (Ed.), *History of Programming Languages I* (pp. 199–243). New York, NY, USA: ACM.

Santori, M. (1990). An instrument that isn't really (Laboratory Virtual Instrument Engineering Workbench). *IEEE Spectrum*, *27*(8), 36–39. doi:10.1109/6.58432

Scholtz, J., & Wiedenbeck, S. (1990). Learning second and subsequent programming languages: A problem of transfer. *International Journal of Human-Computer Interaction*, *2*(1), 51–72.

ScratchBlocks. (2014). Retrieved from https://github.com/blob8108/scratchblocks2

Scribner, S., & Cole, M. (1981). *The psychology of literacy* (Vol. 198). Harvard University Press Cambridge, MA.

Sengupta, P., & Wilensky, U. (2009). Learning Electricity with NIELS: Thinking with Electrons and Thinking in Levels. *International Journal of Computers for Mathematical Learning*, *14*(1), 21–50.

Sherin, B. L. (2000). How students invent representations of motion: A genetic account. *The Journal of Mathematical Behavior*, *19*(4), 399–441.

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

Sherin, B. L. (2001b). How students understand physics equations. *Cognition and Instruction*, *19*(4), 479–541.

Sherin, B. L., Reiser, B. J., & Edelson, D. (2004). Scaffolding analysis: Extending the scaffolding metaphor to learning artifacts. *Journal of the Learning Sciences*, *13*(3), 387–421.

Slany, W. (2014). Tinkering with Pocket Code, a Scratch-like programming app for your smartphone. In *Proceedings of Constructionism 2014*. Vienna, Austria.

Smith, D. C. (1977). *Pygmalion: A Computer Program to Model and Stimulate Creative Thought*. Birkhäuser.

Smith, D. C., Cypher, A., & Spohrer, J. (1994). KidSim: Programming agents without a programming language. *Communications of the ACM*, *37*(7), 54–67.

Smith, D. C., Cypher, A., & Tesler, L. (2000). Programming by example: novice programming comes of age. *Communications of the ACM*, *43*(3), 75–81.

Soloway, E., Guzdial, M., & Hay, K. E. (1994). Learner-centered design: The challenge for HCI in the 21st century. *Interactions*, *1*(2), 36–48.

Sorva, J. (2012). *Visual Program Simulation in Introductory Programming Education*. Aalto University, Espoo, Finland.

Stead, A., & Blackwell, A. F. (2014). Learning Syntax as Notational Expertise when using DrawBridge. Presented at the Psychology of Programmign Interest Group, University of Sussex.

Stefik, A., & Siebert, S. (2013). An Empirical Investigation into Programming Language Syntax. *ACM Transactions on Computing Education*, *13*(4), 1–40. doi:10.1145/2534973

Stieff, M., & Wilensky, U. (2003). Connected chemistry—incorporating interactive simulations into the chemistry classroom. *Journal of Science Education and Technology*, *12*(3), 285–302.

Swetz, F. (1989). *Capitalism and arithmetic: The new math of the 15th century*. La Salle, Illinois: Open Court.

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. *Education, IEEE Transactions on*, *53*(1), 53–60.

Taylor, C., Zingaro, D., Porter, L., Webb, K. C., Lee, C. B., & Clancy, M. (2014). Computer science concept inventories: past and future. *Computer Science Education*, *24*(4), 253–276.

Teitelbaum, T., & Reps, T. (1981). The Cornell program synthesizer: a syntax-directed programming environment. *Communications of the ACM*, *24*(9), 563–573.

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

Tew, A. E., & Guzdial, M. (2010). Developing a validated assessment of fundamental CS1 concepts. In *Proceedings of the 41st ACM technical symposium on Computer science education* (pp. 97–101).

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

Tynker. (2014). Mountain View, CA: Tynker. Retrieved from http://www.tynker.com/

Van Deursen, A., Klint, P., & Visser, J. (2000). Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*, *35*(6), 26–36.

Vygotsky, L. (1978). *Mind in society: The development of higher psychological processes*. (M. Cole, V. John-Steiner, S. Scribner, & E. Souberman, Eds.). Cambridge, MA: Harvard University Press.

Vygotsky, L. (1986). *Thought and language*. Cambridge, MA: MIT Press.

Wagh, A., & Wilensky, U. (2012). Evolution in blocks: Building models of evolution using blocks. In C. Kynigos, J. Clayson, & N. Yiannoutsou (Eds.), *Proceedings of the Constructionism 2012 Conference*. Athens, Greece.

Weintrop, D., & Wilensky, U. (2012). RoboBuilder: A program-to-play constructionist video game. In C. Kynigos, J. Clayson, & N. Yiannoutsou (Eds.), *Proceedings of the Constructionism 2012 Conference*. Athens, Greece.

Weintrop, D., & Wilensky, U. (2014). Situating programming abstractions in a constructionist video game. In G. Futschek & C. Kynigos (Eds.), *Proceedings of Constructionism 2014*. Vienna, Austria.

Werner, L., McDowell, C., & Denner, J. (2013). A first step in learning analytics: pre-processing low-level Alice logging data of middle school students. *Journal of Educational Data Mining*, *5*(2), 11–37.

Wertsch, J. V. (1991). *Voices of the mind: A sociocultural approach to mediated action*. Cambridge, MA: Harvard University Press.

Whitley, K. N. (1997). Visual programming languages and the empirical evidence for and against. *Journal of Visual Languages & Computing*, *8*(1), 109–142.

Whorf, B. L., Carroll, J. B., & Chase, S. (1956). *Language, thought, and reality: Selected writings of Benjamin Lee Whorf*. MIT press Cambridge, MA.

Wiedenbeck, S. (1993). An analysis of novice programmers learning a second language. In *Empirical Studies of Programmers: Fifth Workshop: Papers Presented at the Fifth Workshop on Empirical Studies of Programmers, December 3-5, 1993, Palo Alto, CA* (p. 187). Intellect Books.

Wilensky, U. (1991). Abstract meditations on the concrete and concrete implications for mathematics education. In I. Harel & S. Papert (Eds.), *Constructionism.* Norwood N.J.: Ablex Publishing Corp.

Wilensky, U. (1995). Paradox, programming, and learning probability: A case study in a connected mathematics framework. *The Journal of Mathematical Behavior*, *14*(2), 253–280.

Wilensky, U. (1997). *StarLogoT*. Center for Connected Learning and Computer-Based Modeling, Northwestern University. https://ccl.northwestern.edu/cm/StarLogoT/.

Wilensky, U. (1999). *NetLogo*. Evanston, IL: Center for Connected Learning and Computer-Based Modeling, Northwestern University. http://ccl.northwestern.edu/netlogo.

Wilensky, U. (2001). Modeling nature's emergent patterns with multi-agent languages. In *Proceedings of EuroLogo* (pp. 1–6). Linz, Austria.

Wilensky, U., & Novak, M. (2010). Teaching and Learning Evolution as an Emergent Process: The BEAGLE project. In R. Taylor & M. Ferrari (Eds.), *Epistemology and Science Education: Understanding the Evolution vs. Intelligent Design Controversy*. New York: Routledge.

Wilensky, U., & Papert, S. (2006). *Restructurations: Reformulations of knowledge disciplines through new representational forms*. (Manuscript in preparation).

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.

Wilensky, U., & 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), 171–209.

Wilensky, U., & Resnick, M. (1999). Thinking in levels: A dynamic systems approach to making sense of the world. *Journal of Science Education and Technology*, *8*(1), 3–19.

Wilkerson-Jerde, M. H., & Wilensky, U. (2010). Restructuring Change, Interpreting Changes: The DeltaTick Modeling and Analysis Toolkit. In J. Clayson & I. Kalas (Eds.), *Proceedings of the Constructionism 2010 Conference*. Paris, France.

Wilson, A., & Moffat, D. C. (2010). Evaluating Scratch to introduce younger schoolchildren to programming. *Proceedings of the 22nd Annual Psychology of Programming Interest Group (Universidad Carlos III de Madrid, Leganés, Spain*.

Wolber, D., Abelson, H., Spertus, E., & Looney, L. (2011). *App Inventor: Create Your Own Android Apps*. Sebastopol, Calif: O'Reilly Media.

Wood, D., Bruner, J. S., & Ross, G. (1976). The role of tutoring in problem solving. *Journal of Child Psychology and Psychiatry*, *17*(2), 89–100.

Yaroslavski, D. (2014). *Lightbot*. Retrieved from http://lightbot.com

Zweben, S., & Bizot, B. (2014). 2013 Taulbee Survey. *COMPUTING*, *26*(5).