

© Copyright 2019

Robert Haven Thompson

HAWK/KW: An Online System for Studying Dyslexic Children's Reading,  
Writing, and Programming

Robert Haven Thompson

A dissertation  
submitted in partial fulfillment of the  
requirements of the degree of

Doctor of Philosophy

University of Washington  
2019

Reading Committee:  
Steven Tanimoto, Chair  
Brian R. Johnson  
Stephen Kerr

Program Authorized to Offer Degree:  
Computer Science and Engineering

University of Washington

**Abstract**

**HAWK/KW: An Online System for Studying Dyslexic Children's  
Reading, Writing, and Programming**

Robert Haven Thompson

Chair of the Supervisory Committee:  
Steven Tanimoto  
Computer Science and Engineering

This thesis describes the design and use of an online system of computerized lessons built to teach reading and writing to students with learning disabilities as part of a large research project on specific learning disabilities in written language. The system comprises two main components: HAWK, the portion that teaches reading and writing skills, and Kokopelli's World, the addition to HAWK that introduces programming instruction. Kokopelli's World is a novel blocks-based coding environment designed to cater to the needs of dyslexic students. The thesis also presents two user studies that explore the relationship between programming and dyslexia. Results suggest that different measures of dyslexia may have different correlation with programming success. The thesis also presents detailed coding experiences of two dyslexic students and discusses them in terms of a newly constructed set of recurring programming patterns. These studies have identified a set of hypotheses about how dyslexic students relate to computer programming that can help structure future research on programming by dyslexic children.



# TABLE OF CONTENTS

1.	Introduction .....	1
1.1	Dyslexia .....	1
1.2	The System.....	1
1.3	Thesis Overview .....	2
2.	Related Prior Work.....	4
2.1	Programming Systems .....	4
2.2	Educational Psychology .....	12
2.3	Why Focus on Dyslexics? .....	14
2.3.1	Dyslexia Measures .....	15
2.3.2	Addressing Dyslexia Through Technology .....	16
2.4	Getting Dyslexics to Participate in Computer Science .....	17
2.4.1	Prior Work with Dyslexia and Programming .....	18
2.4.2	Why Study Dyslexic Children? .....	19
3.	HAWK .....	21
3.1	Design Process.....	23
3.1.1	Design Rationale.....	24
3.1.2	User Progression Through HAWK .....	25
3.2	Results.....	35
3.3	Conclusion.....	37
4.	Kokopelli's World Design .....	38
4.1	Kokopelli's World System Components.....	38

4.2 Kokopelli's World Features and Design Rationale.....	40
4.3 Kokopelli's World and Dyslexics.....	45
4.4 Kokopelli's World Block Descriptions.....	47
4.4.1 Subjects .....	48
4.4.2 Objects.....	49
4.4.3 Verbs .....	51
4.4.4 Other Blocks .....	53
4.5 List of KW Lesson Types.....	56
4.5.1 Introductory lesson.....	56
4.5.2 Lessons introducing system concepts.....	56
4.5.3 Lessons with loops .....	57
4.5.4 Lessons with Conditionals.....	58
4.5.5 Lessons about Nested Control Structures .....	60
4.5.6 Lessons about Functions.....	61
4.5.7 Lessons about Randomness & End-User Input.....	62
4.6 Differences Across KW Versions.....	63
5. Experiments, Analysis and Results.....	67
5.1 Data Format.....	67
5.2 Experiment 1: KW HAWK.....	69
5.3 Experiment 2: KW Class.....	73
5.3.1 Reading Test .....	74

5.3.2 Time Taken.....	78
5.3.3 Qualitative Results .....	81
5.4 Student Code Experiences.....	86
5.4.1 Coding Patterns.....	87
5.4.2 Pattern Rationale.....	90
5.4.3 Aggregate Pattern Counts.....	92
5.4.4 Detailed Student Code Experiences .....	93
5.4.5 Individual Student Tallies.....	118
5.4.6 Interview Responses.....	120
6. Discussion .....	126
6.1 Reflections on the KW HAWK Results.....	126
6.2 Reflections on the KW Camp Results.....	127
6.3 Reflections on the KW Class Results .....	128
6.4 Difficulty of Testing Target Population.....	132
6.5 Research Hypotheses .....	132
7. Future Work .....	134
7.1 Future Changes to Kokopelli's World .....	134
7.2 Learning Disabilities and Coding .....	136
7.3 A Possible Model of Dyslexia-Programming Interaction .....	138
7.4 Coding and Literacy.....	140
8. Conclusion .....	142

## LIST OF FIGURES

Figure 3.1: A diagram of the main components of the HAWK system and their interactions.....	22
Figure 3.2: Index of lessons from the HAWK 5 version. ....	26
Figure 3.3: Screen from the Letters in Motion section of HAWK. ....	27
Figure 3.4: Screen from the Words in Motion section of HAWK.....	27
Figure 3.5: Screen from the Minds in Motion section of HAWK. ....	28
Figure 3.6: Screenshot of the Letters in Motion lesson. ....	29
Figure 3.7: Screenshot from the Cross Code Talking-Writing Matrix activity.....	30
Figure 3.8: Screenshot of the Connecting Words through the Musical Melody of Language activity.....	31
Figure 3.9: Screenshot of the Sentence Word Order activity.....	32
Figure 4.1: A diagram of the core components of Kokopelli's World and how they interact. ....	39
Figure 4.2: Similar commands in KW (top) and Scratch (bottom) ....	41
Figure 4.3: Popular silhouette depiction of Kokopelli.....	42
Figure 4.4: Kokopelli's World interface with the microworld visible on the right side. ....	43
Figure 4.5: A block in KW with the audio output icon visible on the left. ....	46
Figure 4.6: A conditional block program.....	46
Figure 4.7: Kokopelli subject block and corresponding microworld icon.....	48
Figure 4.8: Raven subject block and corresponding microworld icon. ....	49
Figure 4.9: Berries object block and corresponding microworld icon.....	49
Figure 4.10: Rock block and corresponding microworld icon.....	50
Figure 4.11: Tree icon. Trees do not have an associated object block. ....	50
Figure 4.12: Drum block and corresponding microworld icon. ....	50
Figure 4.13: The Home icon modeled after a Native American longhouse.....	51
Figure 4.14: The conditional block. ....	53
Figure 4.15: The two condition blocks. ....	54



Figure 4.16: The loop block. ....	54
Figure 4.17: The Forever loop block. ....	54
Figure 4.18: The direction block. ....	55
Figure 4.19: The end block. ....	55
Figure 4.20: A student-defined function definition block.....	56
Figure 4.21: Activity in KW teaching the student how to connect blocks horizontally. ....	57
Figure 4.22: Activity in KW introducing the students to the loop block.....	58
Figure 4.23: Activity in KW letting users try if blocks for the first time .....	60
Figure 4.24: This KW activity introduces nested control structures. ....	61
Figure 4.25: KW HAWK activity teaching functions. ....	62
Figure 4.26: The final KW HAWK lesson that incorporates end-user input and randomness. ....	63
Figure 5.1: A screenshot of a single question from the second half of the KW reading test. ....	76
Figure 5.2: Histograms of the student reading scores.....	77
Figure 5.3: Boxplots for the time students took in seconds (Y axis) to complete each lesson (X axis). ....	79
Figure 5.4: Student Dave’s attempt to create an incorrect command.....	94
Figure 5.5: Dave’s program when he tried to make a second command “verb-first”.....	95
Figure 5.6: Student Dave’s example of the subject-stack pattern. ....	96
Figure 5.7: Dave’s attempt to solve lesson 14 .....	97
Figure 5.8: Dave’s attempt to get Kokopelli to dance three times each time after he moves. ....	98
Figure 5.9: A program from Dave’s attempt at solving lesson 22 .....	99
Figure 5.10: Another snapshot of Dave solving lesson 22.....	100
Figure 5.11: Another program from Dave’s attempt at solving lesson 22.....	101
Figure 5.12: Another common incorrect solution for lesson 22 demonstrated by Dave. ....	102
Figure 5.13: Dave’s incomplete solution for lesson 23. ....	103
Figure 5.14: John’s program missing a drop object block.....	105
Figure 5.15: John’s program with commands in the incorrect location (outside of the loop). ....	106

Figure 5.16: John's program where he makes all the verbs he intends to use in advance. ....	107
Figure 5.17: John's incorrect solution to lesson 17 where he expresses frustration. ....	107
Figure 5.18: Another incorrect solution for lesson 17. ....	108
Figure 5.19: John's first attempt at lesson 18. ....	109
Figure 5.20: Sarah's program attempting to connect two verbs together in lesson five. ....	112
Figure 5.21: Sarah's more complex verb stacking program from lesson nine. ....	112
Figure 5.22: Sarah's program with a correctly-structured loop and superfluous commands afterwards..	113
Figure 5.23: Sarah's lesson 13 program with repeated commands that still passes. ....	113
Figure 5.24: Sarah's lesson 14 program which uses a loop, but does so incorrectly. ....	114
Figure 5.25: Sarah's lesson 16 program where she ignores the required conditional. ....	115
Figure 5.26: Sarah's peculiar pattern from lesson 18. ....	116
Figure 5.27: Sarah's program from lesson 19. ....	117
Figure 0.1: Writing feedback for a drawn letter 'O' using the length method. ....	149
Figure 0.2: Writing feedback for a drawn letter 'O' using the time method. ....	150
Figure 0.3: Writing feedback for a drawn letter 'O' using the length+time method. ....	151
Figure 0.4: Writing feedback for a drawn letter 'O' using the ductus method. ....	152
Figure 0.5: Questionnaire given to students during week eight of the KW HAWK study. ....	157
Figure 0.6: Questionnaire given to students during week nine of the KW HAWK study. ....	158
Figure 0.7: Questionnaire given to students in an interview format during week ten. ....	159
Figure 0.8: Screenshots of the KW Class questionnaire filled out by students. ....	161

## LIST OF TABLES

Table 4.1: A table specifying the feature differences between the three versions of KW. ....	64
Table 5.1: Aggregate correct/incorrect usage statistics for control blocks. ....	71
Table 5.2: Coding success and fluency per student participant.....	72
Table 5.3: The number of observed occurrences of the previously described code patterns. ....	92
Table 5.4: Coding pattern tallies for the three featured students.....	119
Table 7.1: Matrix of possible interactions between the strengths and weaknesses of dyslexics.....	139
Table 0.1: ANOVA analysis results comparing raw time taken to complete lessons. ....	155
Table 0.2: ANOVA analysis results comparing normalized time taken to complete lessons.....	155
Table 0.3: ANOVA analysis results comparing student reported affect scores. ....	156

# **ACKNOWLEDGEMENTS**

Thank you to Dr Steve Tanimoto, who has been my unfailing adviser, mentor, advocate and friend for nearly ten years now. Thank you to my parents and Janny for your absolute confidence that I would succeed, even when I did not share the sentiment. Thank you to Dr. Virginia Berninger for providing me with a much-needed perspective from outside of the CS space, and for providing the opportunity to begin work on this topic. Thank you also to Sandra and Tyler for providing early, key examples of what successful and kind grad students can be.

# 1. Introduction

## 1.1 Dyslexia

Dyslexia is a learning disability that affects millions of people across the world. It manifests in people as a reading difficulty that affects the speed and accuracy of word decoding (Siegel 2006). Dyslexia can also affect brain function related to reasoning and memory in ways that are not yet fully understood or defined. Dyslexia can be difficult to diagnose since it is based on performances in reading tests with continuous scores. By definition any cutoffs used for a diagnosis are arbitrary. Dyslexia does not affect general intelligence but can hinder the educational growth of students by preventing them from reading at the same level as their peers (Castro et al. 2018). For several decades now, research projects have been conducted to study how computerized lessons providing instruction can be used to improve the learning experience of dyslexics, particularly dyslexic children (Hall et al. 2000; Seo and Bryant 2009). The vast majority of this work up to this point has focused on reading and writing tasks. Programming is an activity that involves both reading and writing but there have been almost no studies examining the potential effects of dyslexia on programmers. This thesis introduces the design and study of a system we have created that supports research on this topic.

## 1.2 The System

The system described in this thesis consists of two parts: “HAWK” and “Kokopelli’s World”. HAWK (Help Agent for Writing Knowledge) is a platform of online computerized lessons built for use on the Apple iPad for children with learning disabilities, including dyslexia, that has shown to significantly improve child performance in reading and writing measures where traditional instruction has failed. HAWK was created by us and others in the Center for Defining and Treating Specific Learning

Disabilities in Written Language as part of a grant from the Eunice Kennedy Shriver National Institute of Child Health and Human Development. This grant supported research on understanding, and treating, learning disabilities with contributions from a variety of academic perspectives including genetics, neurology, educational psychology, and computer science. HAWK's role in this task was to be a platform for administering computerized instruction and measuring its effects on student learning.

Near the end of HAWK's grant's lifespan, we expanded the system's original range of topics from just reading and writing to also include programming. We developed an additional computer subsystem to support this research: Kokopelli's World (KW). Since then, KW as helped us carry out some of the earliest work examining the interactions between dyslexia and programming. More study will be required, but results from work with KW suggest dyslexia may not be a strong barrier for young programming learners. Related work and theoretical models suggest in fact that dyslexic individuals may even be well-suited to some programming tasks. This is encouraging initial evidence that a significant portion of the population may not be hindered in participating in one of the fastest-growing and important fields of the modern age.

### 1.3 Thesis Overview

The structure of this thesis is as follows. First we give a brief summary of the history of educational programming environments as they relate to KW, as well as a formal definition of learning disabilities and dyslexia for those less familiar with the topic. Then we present HAWK's design and a summary of results from several research studies. We follow that with a detailed examination of Kokopelli's World's design and the motivation behind it. We then describe and analyze the results from two studies related to programming that occurred with KW and HAWK. Finally, we end with a discussion of the future of KW's work and dyslexia's relationship to programming as a whole.

This thesis presents four primary research contributions:

- The design of HAWK, a novel set of computerized lessons for teaching reading and writing to dyslexic children
- Several studies involving HAWK that explore and answer numerous research questions around learning-disabled instruction
- The design of Kokopelli's World, a novel programming environment built to assess and address the needs of dyslexic programming learners
- Two studies with KW examining, for the first time, the potential effects of dyslexia on children learning to program

## 2. Related Prior Work

The designs of Kokopelli's World and HAWK have drawn influence and inspiration from computer science as well as educational psychology research. This chapter discusses related prior work from both of these fields. We first give a brief history of related programming environments, including novice programming systems, blocks-based environments, and storytelling programming tools. We then define dyslexia and discuss the history of dyslexia research as it relates to computers and programming specifically. We end this chapter with a discussion of why dyslexia research in the computer science field is relevant and important.

### 2.1 Programming Systems

Educational programming systems began early in the history of programming. Languages such as BASIC from the early 60s were attempts to create programming systems that were more accessible to a broader audience not limited to professional programmers (Kemeny and Kurtz 1964).

Naturally, these initial attempts had their limitations. BASIC was still a language represented entirely in text. As short as a few years later, systems begin to appear that explicitly target children like Logo and its related project TORTIS (Abelson 1974; Morgado 2006). Visual systems were introduced in the same decade with Ivan Sutherland's Sketchpad (Sutherland 1963). By the 80s, we see the rise of the personal computer, and with it a much wider potential audience for programming systems to target. Commercial projects such as Story Machine appear (Infoworld 1983). In the research field, a focus on programming metaphors grows with projects such as Programming by Rehearsal and PLAY (Gould and Finzer 1984; Tanimoto and Runyan 1986). With the 90s, blocks programming comes into the fore as projects like ALICE and AgentSheets debut (Conway et al. 1994; Reppening 1991). This trend continues into the 2000's as more modern projects like Scratch



and App Inventor focus on social aspects, offer real-world practicality, and support an unprecedented number of users in the millions (Resnick et al. 2009; Hardesty 2010; Gray et al. 2012).

In this section we will briefly review the history of educational coding environments relevant to Kokopelli's World including the projects mentioned above and more. This is by no means intended to be a comprehensive list, but rather a representative sample of the numerous projects that in some way informed the design of our own system. This is also useful in delineating what makes KW different from previous similar systems.

### **Sketchpad**

Sketchpad, developed by Ivan Sutherland in 1963 for his PhD thesis (Sutherland 1963), was a program for drawing. It was a seminal project that paved the way for whole fields of computing work such as human computer interaction (HCI), Computer Aided Design (CAD), and graphical user interfaces (GUIs). For the purposes of this thesis, it serves as the origin point of both graphical systems, as well as end-user systems, predating even the BASIC language that would debut the following year.

### **BASIC**

Beginner's All-Purpose Symbolic Instruction Code (BASIC), developed by Kemeny and Kurtz and first released in 1964, is generally agreed upon to be the first educational programming language (Kemeny and Kurtz 1964). One thing that set it apart from previous languages was its intended audience. BASIC was the first language designed to be usable by non-programmers. With the rise of the IBM personal computer after 1981 it became popular with hobbyists and others who we would later come to call end-user programmers.

## **Logo**

Logo was the brainchild of Feurzeig and Papert, originally released in 1967, and meant to teach LISP programming concepts to young children by letting them construct commands that resembled natural language sentences (Abelson 1974). A few years after its original release Papert would add a graphical component to Logo to create Turtle Graphics, now the more commonly used version of Logo (Thornburg 1983). With Turtle Graphics, users control an onscreen turtle that can trace its path of movement as it follows commands. With this trace, users can create images. Papert would later define the concept of body-syntonicity, claiming that users had an easier time reasoning about the turtle's path of movement when they placed their perspective inside of the turtle. This concept relates to the use of a microworld by later systems, including Kokopelli's World. Before Logo was formally released, a different implementation of it, with a physical turtle robot and arguably one of the earliest blocks-based programming languages, was being developed in the same lab by Radia Perlman.

## **TORTIS**

TORTIS was developed during 1974-1976. Radia Perlman created TORTIS as a tangible programming system that grew out of the Logo project (Morgado 2006). Logo itself was a children's programming language by Papert and others but required some ability to read (Papert and Solomon 1971; Papert 1980). In TORTIS, children built programs out of physical blocks that would then control a robot that could move forward, backward or rotate, lower and raise a pen, toggle a light, and beep. The TORTIS environment supported command repetition and up to four user-defined functions. Because TORTIS was entirely icon-based, reading skill was not required, and children as young as three used and enjoyed the system.

## **Story Machine**

While Story Machine was a commercial endeavor rather than a research project, and not a programming language, one could consider it a precursor to story-focused blocks-based systems

like Scratch. Story Machine was released in 1982 by Spinnaker Software (Infoworld 1983). It allowed users to construct simple sentences from a predefined set of words. These sentences would then be “acted out” by the computer in a simple animation for the user to see.

The notion of using natural language to instruct computers dates at least as far back as 1966, when Jean Sammet discussed both the potential use and challenges of English as a computer interface language (Sammet 1966). Sammet mentions several ideas that would continue to be relevant to the field for decades: the difficulty in creating a programming language that mimics English, the different but equal difficulty of unambiguously interpreting natural English, and the importance of voice in the future.

### **Programming By Rehearsal**

Programming By Rehearsal (Gould and Finzer 1984) was a visual programming environment to allow non-programmers to create educational software. Programming By Rehearsal framed the programming process as a play, where “performers” acted out on “stages” and interacted through different “cues”. One key focus of the system was to represent every state of the program visually.

### **PLAY**

Pictorial Language for Animation by Youngsters (PLAY) was an iconic, text-free programming system for early PCs (Tanimoto and Runyan 1986). Programs were referred to as “scripts” and represented like a comic strip. Commands were created by user demonstration and each involved a subject, verb, and adverb. Like Programming By Rehearsal, PLAY presented theater as a metaphor for programming with a microworld as the stage and the user as a playwright or director.

### **AgentSheets**

AgentSheets is a programming environment developed by Alex Repenning and originally released in 1991 (Repenning 1991). AgentSheets pioneered several features of a novice programming environment that would become popular with later systems, such as supporting an online

community and rule-based programming. Rather than attempting to be a general coding or storytelling environment, AgentSheets focuses on offering users tools to create scientific simulations appropriate for primary school students to make and understand. Agent Cubes is an offshoot of AgentSheets that supports a 3D environment.

### **Alice**

Alice is a blocks-based programming language tied to a 3D microworld that allows users to create and manipulate 3D actors (Conway et al. 1994). It was developed by a team of researchers led by Randy Pausch. Alice has a strong emphasis on storytelling and targets younger students, typically around middle school age.

Blocks languages eliminate the need for most typing, and as such they can present a unique benefit for users with difficulty spelling, including dyslexics. Alice is one of a small set of languages that has been studied with dyslexics and found to produce positive results in comparison to text-based languages (Castro et al. 2018).

### **Comic Chat**

Comic Chat is an instant messaging client developed by Microsoft in 1996 (Kurlander et al. 1996). The goal of the project was to take natural dialog between two or more people in a chat client and create a multi-frame comic using that text. Comic Chat is not a programming language but had to address many of the same issues as other natural language oriented tools through its development, such as understanding tone and context and resolving ambiguities.

### **Toon Talk**

ToonTalk is another example of a system designed to be usable by pre-literate children (Kahn 1996). ToonTalk provides a constraint-based system and presents itself like an open-ended Lego sandbox. Children program robot-like agents by inserting blocks, all represented by icons instead of text, in to a work space and programming by demonstration. While ToonTalk could in principle be

used to tell stories, it presents itself more as a game where users can manage a town and create their own games to share with friends.

### **Metafor**

Metafor is a “brainstorming” editor for automatically converting English prose to partially specified code scaffolding, namely Java interfaces and classes (Liu and Lieberman 2005). It is meant to be used as an outlining tool for coding projects. Metafor models language through a method they call programmatic semantics, which dictates how natural language is interpreted into code.

### **Scratch**

Scratch is a system developed in the MIT Media Lab’s Lifelong Kindergarten Group, led largely by Mitch Resnick (Resnick et al. 2009). Like its predecessors, Scratch uses blocks to abstract more complex programming syntax and make errors related to that syntax impossible. Like Alice, Scratch uses an ever-present environment, or microworld, in which actors perform actions based on the user’s program. Scratch differentiates itself from other blocks-based systems with its vibrant online community that hosts a forum and simple tools for sharing and ‘remixing’ code. The system itself also supports optional user-created libraries and modifications, further stretching its functionality. As of July 2016 the online Scratch community had more than 12 million registered users who had made more than 15 million code projects.

### **Gidget**

Gidget comes out of the work of Michael Lee (Lee and Ko 2011). Gidget is a set of debugging challenges involving fixing text code to make a robot correctly perform a series of tasks within a microworld. Lee’s focus for the software was to explore more helpful, human-friendly debugging feedback methods in the context of a pre-defined set of puzzles and challenges, rather than a completely open environment.

## **App Inventor**

MIT App Inventor was developed largely by Abelson and Friedman at Google in 2010 (Hardesty 2010; Gray et al. 2012). A second version, now the one most-used, was created at the MIT Media Lab. App Inventor shares a similar interface and style as Scratch but focuses on providing tools to create mobile applications. This fact alone makes it popular among younger audiences though it arguably has a higher barrier to entry than Scratch.

The current version of AppInventor uses Google Blockly to create and represent blocks on the client-side. Blockly is not a blocks-based language itself, but rather a library that allows others to easily create their own blocks-based languages (Fraser 2012). Kokopelli's World also uses a modified version of Blockly.

## **Scratch Jr.**

Scratch Jr. is an attempt to adapt the blocks-based Scratch programming environment to K-2 students (Flannery et al. 2013; Resnik et al. 2009). It focuses primarily on story creation. Scratch Jr.'s design goals were to offer an environment with a low floor and a high ceiling that can engage students at a young age but still support creative solutions as more complex languages do.

Programming in Scratch Jr., as in Scratch, involves connecting blocks together to form sequential commands. Blocks' labels are icons, to enable pre-literate users to interact with the system. Most commands involve the manipulation of one of several on-screen 2D sprites, and nearly all have immediately visible effects. These sprites represent the characters children use to tell their stories. Stories/programs can also be shared between students, and Scratch Jr. offers classroom support for teachers to facilitate multiple students using the language simultaneously.

## **FLIP**

FLIP is a visual scripting language for beginners to create scenarios in the Neverwinter Nights 2 scenario creation toolset (Howland and Good 2015). Their design was informed by earlier studies examining the viability of natural language as a programming language. FLIP presents the current

program state in two variations: one as a visual blocks system, and the other as a natural-language interpretation of the code. This has some similarity to the natural language audio output systems used in KW.

The languages, environments, and systems mentioned above represent several design themes present in Kokopelli's World as well. The first and broadest is the educational programming environment. These are programming environments developed primarily for their ability to teach or inspire users rather than for the breadth of programs that can be created from them. These types of environments often target children specifically as well, and KW is no exception. Educational environments can support complex programs and projects, but many of them are meant to be stepping stones in a programmer's personal growth; a place for them to experiment and explore programming concepts that will be applicable in other, more advanced environments later on. As the movement to teach younger children programming grows in public schools, these sorts of environments are becoming ever more important as educational tools.

Many of these environments make use of visual programming languages. Visual languages can offer a host of benefits for programming students over textual ones. For instance, they can be a blocks programming environment that lessens or eliminates syntax errors, or offer a microworld with characters that have implied semantic behavior. If students are shown a character in a 2d, top-down view of a microworld they can assume that that character will be able to move through the space in some way, and interact with other objects in the world. Because students can predict the types of commands that are allowed, they can spend more time on other aspects of programming.

Some environments seek to present another activity as a metaphor for programming. This can be used to interest a potential user or ground a set of programming actions and constructs that otherwise might come across as arbitrary or overly abstract. The PLAY and Programming By Rehearsal environments present theater as a metaphor for programming, while Alice and FLIP emphasize programming more broadly as a form of storytelling. In addition to being an effective

metaphor for programming, storytelling provides unique benefits as a programming activity. It is culturally universal and disconnected from technology in a way that can interest users who might not be motivated to make a game or app.

Storytelling leads to one final theme: that of using natural language in programming. The environments above show the range of ways natural language can be applied to programming. Firstly, there is the idea of using natural language as input for a computer to generate code or behavior. The Metafor system and arguably Comic Chat and Story Machine could be considered examples of this. Taken more broadly, many text languages that adopt a more intuitive syntax structure are attempting this as well. Logo was never presented as a system that could be misinterpreted as natural language, but it was an attempt to create a language with more natural-language-like syntax rules to make it more approachable for programming novices. Other environments have taken the opposite approach, and have the computer try to generate natural language to more easily communicate program state or behavior to the user. The FLIP environment is one example of allowing users to see their code framed as natural language.

## 2.2 Educational Psychology

This thesis assumes the reader has a computer science background. As such, this section will briefly define learning disabilities for individuals unfamiliar with them.

The US National Institute of Child Health and Development defines learning disabilities broadly as conditions that “affect how a person reads, writes, speaks, or calculates” and that they are caused by “differences in a person’s brain”. It is still unknown what precisely causes individuals to develop learning disabilities, though it appears to be a complex combination of genetic, neurological and environmental factors. Learning disabilities do not negatively affect general intelligence (US NICHD 2018). They are not a form of physical disability, such as hearing or motor control loss, nor a form of mental retardation.



“Learning disability” is a blanket term that refers to a number of more specific conditions. There is still disagreement about which conditions are formal learning disabilities. Some scientists include attention deficit hyperactive disorder and/or autism as learning disabilities while some do not, for example, and the precise definition of each particular condition is still a subject of debate. As such, it is difficult to gauge how many individuals have some form of learning disability. In the US 2010 census, 3.9 million or 1.6 percent of the population reported having a learning disability (Brault 2010). However, this is self-reported and may be mis-representative. Some other estimates include: 6.9-8.2 percent of US children have at least one specific learning disability (Halfon et al. 2012); 10 percent of the population have a learning disability, including 4-8 percent dyslexia (Butterworth and Kovas 2013); 5-17 percent of school-aged US children have dyslexia (Shaywitz and Shaywitz 2005).

The most commonly diagnosed learning disability is dyslexia. Dyslexia is a neurological condition that impairs a person’s ability to read. Dyslexia is actually an umbrella term that describes several specific impairments that may occur individually or in combination in an individual. Some of these conditions include difficulty matching letters to sounds, difficulty reading quickly and accurately, poor reading comprehension, and poor spelling. Dyslexia is the focus of this study, but there are many other recognized learning disabilities including dysgraphia (trouble writing), dyspraxia (trouble with motor control), dyscalculia (trouble with mathematics), auditory processing disorder (trouble understanding speech), apraxia (trouble speaking), and non-verbal learning disabilities (trouble with non-verbal communication).

It is also worth mentioning ADHD as part of this discussion. Attention deficit hyperactive disorder (ADHD) is sometimes though not always treated as a learning disability but often co-occurs in children with learning disabilities. Children with dyslexia have been found to be 30% likely to also have ADHD (International Dyslexia Association 2017). As such, when considering systems that address the needs of learning-disabled children one should also consider the potential for them to have ADHD. Individuals with ADHD often have difficulty focusing on one task for

longer periods of time, may be unable to stay still for long periods of time, and can act impulsively before considering their actions. For many people, ADHD can be treated with medication, unlike with learning disabilities.

## 2.3 Why Focus on Dyslexics?

As described above, learning-disabled (LD) individuals can exhibit a wide range of behaviors, some completely orthogonal to each other and some even directly opposed. It is likely not possible to create a static system that caters to all disabilities at the same time. For this research we chose to focus on dyslexic individuals for several reasons:

1. Dyslexia is the most common learning disability throughout the population, though this may primarily be because it is the most well-known and studied. A system for dyslexics benefits the most people out of all learning disabilities.
2. Some research suggests that dyslexics may be well-suited to programming as adults if they are able to overcome the initial hurdles of learning the domain (Powell et al. 2004).
3. We ourselves had prior experience developing reading and writing software for dyslexics and dysgraphics with HAWK that we were able to bring to bear while performing this research.

It is largely outside the scope of this study, but we suspect a system designed for children with other types of learning disabilities could look very different from our own. Auditory processing disorder, for example, affects a child's ability to hear and differentiate sounds and a system that caters to children with it would be unlikely to employ as much audio support as Kokopelli's World does. It would by no means be less useful to consider support for other

disabilities though, and we encourage others to explore the potential for these systems. For those interested, this topic is discussed more in the future work section.

### *2.3.1 Dyslexia Measures*

There is no single, universal test for dyslexia (US NICHD 2018). It can manifest in a variety of ways, some of which are orthogonal to each other. Instead, there is a history of measures that, in the past, have proven to differentiate dyslexics from typically developing individuals. A sample of these tests are listed here to convey in more detail the type of tasks dyslexics struggle with beyond simply “reading.” If the testee distinguishes themselves in a number of these tests - typically if they are in a low percentile - when administered by a trained professional, then they are considered to have dyslexia. Importantly, these are all continuous measures so the exact threshold that separates dyslexic and typically developing will always be somewhat arbitrary. The measures in general usually concern skills related to word reading and spelling, some of which are described below.

#### **Test of Orthographic Competence (TOC)**

This is itself a battery of tests. One task involves asking the testee to choose one of four available letters to fill in the blank of an incomplete word (TOC Letter Choice). In another, the task is to write in missing letters of words after listening to dictated words (TOC Sight Spelling Test) (Mather et al. 2008).

#### **Wechsler Individual Achievement Test, 3rd Edition (WIAT 3) Spelling**

Testees must spell words that are said out loud alone, in a sentence, and then alone again (Pearson 2009).

#### **WJ3 Writing Fluency**

Participants are given three words and must write a complete sentence incorporating each word without changing it (Woodcock et al. 2001).

### **WJ3 Word Attack**

Participants are given a set of fake words - words that follow English construction rules but do not have an associated definition or history of use - and must try to guess how they are spoken following the normal rules of English (Woodcock et al. 2001).

### **Test of Word Reading Efficiency (TOWRE)**

This test measures accuracy as participants pronounce a list of written words while under a time constraint. A variation includes non-English words that are still pronounceable (Torgesen et al. 1999).

### **Test of Silent Word Reading Fluency (TOSWRF)**

Testees must mark word boundaries in a series of letters grouped into rows. The score is how many words are marked in a given span of time (Mather et al. 2004).

### **PAL Sentence Sense Accuracy**

The subject must choose which of three given sentences is a real, meaningful sentence. The other two sentences are correctly constructed but do not make sense in context (Berninger 2008).

### *2.3.2 Addressing Dyslexia Through Technology*

Before we focus in on dyslexia and programming, it is worth mentioning the ways technology has been used to help dyslexics in other subjects.

There is a history of using computers to offer sometimes-effective instruction to children with learning disabilities for subjects such as reading (Hall et al. 2000) and mathematics (Seo and Bryant 2009) dating at least as far back as the early 80's. These indicate that students diagnosed with dyslexia can perform at least at average levels in other fields when provided with proper computerized instruction.

Outside of a lab setting, what technology tools are dyslexics actually using in real life? A survey in Britain of dyslexic students under the Disabled Students Allowance looked at what tools were being given to post-secondary students after their needs had been individually assessed (Draffan 2007). Hardware included recording devices, handheld spell-checkers, smart pens, and portable note-takers. Software included text-to-speech programs most importantly, but also talking dictionaries, concept mapping software, and word-prediction features. The single task of programming is one that occurs largely in the software space. As such, special hardware like recording devices and smart-pens may not be as relevant to dyslexic programmers but the software may be quite beneficial.

## 2.4 Getting Dyslexics to Participate in Computer Science

While there is a detailed, if still ongoing, understanding of the mechanics of dyslexia and how its effects can be addressed through technology in other fields, we are unfortunately not so lucky in regards to computer science. To our knowledge, there have been very few studies that relate dyslexia and computer science at all and none that attempt to directly measure any potential interactions between dyslexia and programming within children.

Considering dyslexics could make up as much as 17 percent of the population, there is value in understanding the effect dyslexia might have on computer programmers. Both dyslexia diagnoses and the number of programming jobs are increasing over time and if dyslexics would benefit from different approaches to teaching computer science than what is currently offered, then it is in society's best interest to discover what those approaches are and begin to utilize them.

While it is not yet a reliable diagnosis tool, studies have shown that fMRI scans of the brain can differentiate LD from typically developing individuals. Furthermore, the brain regions most often associated with dyslexia (Norton 2014) have some overlap with the regions activated during coding comprehension activities (Siegmund 2014); the inferior parietal arena and inferior frontal

gyrus in particular. There are still very few studies that link programming and the brain, but this is early evidence to further suggest that dyslexia may have a significant impact on programming ability.

#### *2.4.1 Prior Work with Dyslexia and Programming*

As mentioned earlier, dyslexics can have difficulty with handwriting, spelling, reading, composition and writing skills, short-term memory, and organization. Of these, handwriting is unlikely to hinder a programmer, but the others have the potential to do so. A 2004 paper by Powell et al. explored the relationship between dyslexia and programming. They theorized that short-term memory and organization would be the biggest obstacles to dyslexic programmers (Powell et al. 2004).

The same study also examined the skills in which dyslexics typically excel. These skills include spatial awareness, visualization, creativity, and lateral thinking, all of which may be important to a programmer. Powell et al. interviewed five adult dyslexic programmers, including students and professionals, about how their disabilities affected their programming skills. Responses varied but the interviewees largely agreed that they saw their disability as an aid to their programming.

Another study by Castro et al. compared the programming output of at-risk-for-dyslexia and not-at-risk adults using visual and text programming languages (Castro et al. 2018). They found that at-risk programmers outperformed the other group and that they were more productive using a visual language over a textual one. This further supports the notion that dyslexics may be well-suited to programming.

A study comparing eye tracking data between dyslexic and non-dyslexic programmers both reading a program found that dyslexics may read code differently than they read text (McChesney and Bond 2018). They also found that dyslexics tend to spend more time looking at other parts of the code than where typically developing programmers focus their attention.

All together, we hypothesize that multiple aspects associated with dyslexia, both positive and negative, could come into play while programming, outside of handwriting. As explained in the next section, our work focuses on children. The first aspect of programming we explicitly teach to beginners is coding. It is also, arguably, the most unique aspect of programming. As such, our own work will focus largely on the coding activity itself.

#### *2.4.2 Why Study Dyslexic Children?*

We determined that children would be ideal research subjects for these tests, at least initially, for a variety of reasons:

- Child dyslexics lack the experience and tools older dyslexic adults have to work around difficult problems
- Children have likely had fewer opportunities to learn programming in their own time, though with the recent movement to introduce programming in primary school this may be changing
- For children that *have* received programming instruction already, the push towards a standardized programming curriculum at least increases the odds that children come with similar past programming experiences
- Children are typically more open to learning new topics
- Children organized into classrooms are a more accessible population for research than adults

Our research focuses on dyslexic children in the elementary and middle-school age range. Children are learning to program at younger ages in greater numbers than before. For example, the Hour of Code initiative, which encourages children who have never programmed before to try it for one hour, is a coding outreach and awareness program started as part of Computer Science

Education Week in 2013. So far more than 100 million students have participated (Hour of Code 2018). Around the world, countries are introducing coding activities into their curricula and at younger and younger ages (Gardiner 2014). It would appear that soon, if not already, children will be expected to have some minimal experience with coding well before high school.

In contrast to this increasing activity, there have as yet been no studies we are aware of looking at dyslexia's effect on children learning to program. If we are to teach these children effectively we need to know if their condition has an effect on their ability to learn programming.



### 3. HAWK

Help Agent for Writing Knowledge (HAWK) is a system comprising a series of online, electronic lessons, and surrounding infrastructure, meant to teach reading and writing to learning disabled (LD) as well as typically developing students. HAWK was developed as part of a project funded by the NIH to further research into learning disabilities occurring in children. The grant included separate but related studies of fMRI scans and gene mapping of blood samples to attempt to identify the root cause(s) and core differentiating characteristics of learning disabilities (Abbot 2017; Richards 2016; Berninger 2015). These were also used as pre and post-tests for a study on the effectiveness of response to instruction on LD children. The HAWK system was designed and created to be the key component of that instruction. HAWK's scope changed and expanded over the course of the five years it was in development and testing, and went through five major iterations, but response to instruction was always its primary purpose.

This context is key to understanding the design constraints on both HAWK and Kokopelli's World, as both were made to fit into this framework. HAWK was intended to be a system that would eventually be able to be run in classrooms for individualized instruction - students each following their own lessons within the system with only minimal help from teachers required. Feedback is largely automatic and comes from the software itself rather than from teachers. Sessions where students used HAWK were about 45 minutes to two hours long and repeated between once per day to once per week for 12 or 18 sessions. The intended audience age range was late elementary and middle school, between the 4th and 9th grades. HAWK is meant to be usable by both typically developing students and learning disabled students with dyslexia, dysgraphia, or the compounded conditions known as Oral and Written Language Learning Disability (OWL LD).

The HAWK system is comprised of several components, most of which are mapped out in Figure 3.1. Most of HAWK exists on a server as a web application. The lesson service component consists of the lesson content and behavior, the database used for user management and data collection, and the controller that manages behavior. The back-end logic of HAWK - for lesson sequencing and site security - is written in PHP using the CodeIgniter framework library and was designed to run on an Apache-based server (EllisLab 2019). HAWK also uses the CommunityAuth library to handle user management (Gottier 2019). The lesson content is defined by a custom-made, JavaScript, client-side code library that dictates the look and behavior of the various HAWK lessons. The content of each lesson changes per session and is specified in a series of static JSON data files that are parsed as needed by the back end. The database is MySQL and stores the two data types key to HAWK: lesson events and lesson scores. Both of these types are discussed in more detail in the KW analysis chapter.

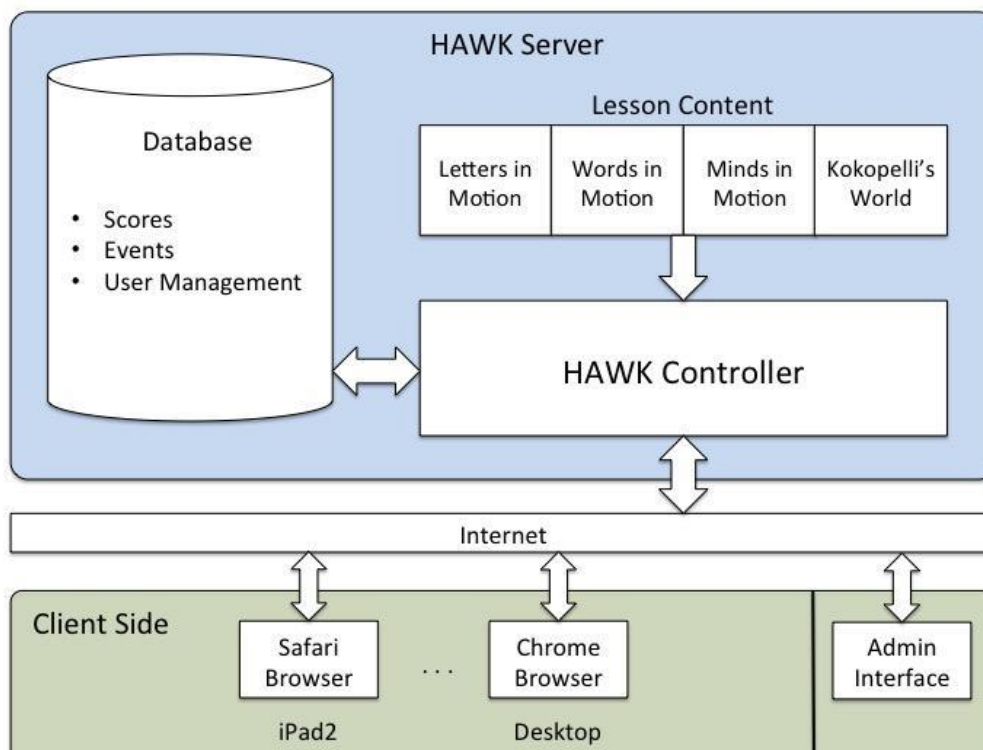


Figure 3.1: A diagram of the main components of the HAWK system and their interactions

The HAWK system can be accessed through any modern device with internet connectivity and a web browser assuming the user has login credentials, though some components of HAWK were designed to be used with audio output and a device with touch or stylus support. HAWK also supports an administrative interface, also accessible through a web browser, for viewing, editing, and outputting collected student data.

The HAWK lesson content is made up mostly of a series of small activities involving either multiple-choice, short answer, or practice drills of various English language theory topics. There are also essay writing portions at the end and, with the addition of Kokopelli's World, coding lessons. Before each lesson students are reminded what they are being tasked to do and, when relevant, are shown a score of how they did in each lesson after it concludes.

Client-side graphics are produced in SVG with the help of the jQuery, Raphael.js, and D3.js libraries. Audio was originally handled with a custom library, but as HTML5 Audio capabilities improved we switched to using the Howler.js library instead.

The rest of this section will cover some of the design rationale behind HAWK, include a more detailed description of some of the lessons that HAWK supports, and briefly summarizes some of the results that came out of the numerous studies involving HAWK.

### 3.1 Design Process

HAWK was created to leverage and study the effects of using computer software to teach lessons on reading and writing for students with learning disabilities. Much of the instructional material was informed by and inherited from the prior work of our collaborator Dr. Virginia Berninger, who had a history of developing pen-and-paper reading and writing instructional materials. The activities in HAWK were previously verified by third parties to be effective in pen-and-paper form (Bear et al. 2000; Henry 2010; Singson et al. 2000; Troia 2009). There were numerous reasons why it was beneficial to incorporate computers into this material. For one, it would likely make the material

more engaging and exciting for students using it. From a teacher's perspective, it meant much easier scale-up to larger classrooms. An electronic system could provide automatic and immediate feedback to all students without any large effort required of the teacher.

From the perspective of the researchers, an online system meant much more data could be collected from users much more easily than with a paper system. New types of data could also be collected, including precise timing information, and new hypotheses could be investigated efficiently, such as the hypothesis that student writing and typing skills are closely connected. Lastly, a server-based computer system would be easier to iteratively improve, as code need only be changed on the central server as opposed to having to re-print any paper materials.

### *3.1.1 Design Rationale*

With these benefits in mind, we had to make several informed platform and format choices early on in the development process. The most important choices are discussed below, but a more detailed discussion of some of the implementation details of HAWK can be found in Appendix B.

#### **Why Online?**

The choice to make the system online was primarily to take advantage of the platform agnostic nature of most websites. With an online system, in theory any web-capable device would be able to access the instructional content, including mobile devices. It also allowed for easy changes to the software, no need to ask users to download a patch, and smoother data storage since the system was already guaranteed to be capable of accessing our data storage server.

#### **Why iPad2s?**

While we intended the system to be accessible from most devices, we still had to choose a specific device to use for testing and with students during studies. At the time (2011), the Apple iPad2 was a device that offered several appealing features. It was popular and would likely be recognized by and

excite students using it. It had input support for both touch by fingers and by stylus, which would allow research comparing them. It also allowed virtual and physical keyboard input, supporting additional research directions. As an Apple product, it offered support and stability while still being cheaper than most laptops. Lastly, while iOS was not as feature-rich as a desktop OS (more on this is discussed in Appendix B), it could run a web browser.

### **Why Scalable Vector Graphics (SVG)?**

Web graphics options in 2011 were more limited without the official release of HTML5 that would happen three years later. We used Scalable Vector Graphics (SVG) as a base for most of our multimedia portions of the HAWK site because it could be created programmatically and remain responsive, unlike static images. It also allowed us to represent writing data in a form that could be re-scaled or analyzed easily.

#### *3.1.2 User Progression Through HAWK*

HAWK was designed to be accessed in spaced, repeated sessions, typically once per week or at most once per day. Each week was considered a lesson with a total of eighteen or, in the offering to one group of students, twelve lessons. Figure 3.2, for example, shows the full list of lessons and activities from the HAWK 5 version, although students would not see this view themselves. Each lesson contains a number of activities, the format for which would repeat each week even as the activity content changed. For example, children would repeat the activity where they count the number of syllables in a word, but every week the words in that activity would change and grow more complex. The broad themes of the activities were divided into three sections. Those sections we called “Letters in Motion”, “Words in Motion”, and “Minds in Motion.” Letters in Motion was primarily focused on teaching and practicing handwriting for individual letters in print and cursive. Words in Motion activities taught reading and writing at the word level and included practice for pronunciation, compound words, anagrams, and much more. Minds in Motion activities involved

either analyzing or composing full sentences and included essay writing sections. HAWK's user interface style changed depending on what section the student was currently in to indicate their progress and emphasize the different modes of thinking required for each. Example screenshots of each mode can be seen in Figure 3.3, Figure 3.4, and Figure 3.5.

<b>Lessons</b>
<b>Letters In Motion</b>
<ul style="list-style-type: none"> <li>Letters in Motion  <a href="#">L1</a>, <a href="#">L2</a>, <a href="#">L3</a>, <a href="#">L4</a>, <a href="#">L5</a>, <a href="#">L6</a>, <a href="#">L7</a>, <a href="#">L8</a>, <a href="#">L9</a>, <a href="#">L10</a>, <a href="#">L11</a>, <a href="#">L12</a> </li> </ul>
<b>Words In Motion</b>
<ul style="list-style-type: none"> <li>Words in Motion Introduction  <a href="#">L1</a> </li> <li>Pattern Analyzer Through Ear  <a href="#">L1</a>, <a href="#">L2</a>, <a href="#">L3</a>, <a href="#">L4</a>, <a href="#">L5</a>, <a href="#">L6</a>, <a href="#">L7</a>, <a href="#">L8</a>, <a href="#">L9</a>, <a href="#">L10</a>, <a href="#">L11</a>, <a href="#">L12</a> </li> <li>Phoneme Pattern Analysis  <a href="#">L1</a>, <a href="#">L2</a>, <a href="#">L3</a>, <a href="#">L4</a>, <a href="#">L5</a>, <a href="#">L6</a>, <a href="#">L7</a>, <a href="#">L8</a>, <a href="#">L9</a>, <a href="#">L10</a>, <a href="#">L11</a>, <a href="#">L12</a> </li> <li>Musical Rhythm of Language Through Stress Patterns  <a href="#">L1</a>, <a href="#">L2</a>, <a href="#">L3</a>, <a href="#">L4</a>, <a href="#">L5</a>, <a href="#">L6</a>, <a href="#">L7</a>, <a href="#">L8</a>, <a href="#">L9</a>, <a href="#">L10</a>, <a href="#">L11</a>, <a href="#">L12</a> </li> <li>Pattern Analyzer for Written Words  <a href="#">L1</a>, <a href="#">L2</a>, <a href="#">L3</a>, <a href="#">L4</a>, <a href="#">L5</a>, <a href="#">L6</a>, <a href="#">L7</a>, <a href="#">L8</a>, <a href="#">L9</a>, <a href="#">L10</a>, <a href="#">L11</a>, <a href="#">L12</a> </li> <li>Combining Two Words to Shorten Them  <a href="#">L1</a>, <a href="#">L2</a>, <a href="#">L3</a>, <a href="#">L4</a>, <a href="#">L5</a>, <a href="#">L6</a>, <a href="#">L7</a>, <a href="#">L8</a>, <a href="#">L9</a>, <a href="#">L10</a>, <a href="#">L11</a>, <a href="#">L12</a> </li> <li>Combining Two Words to Lengthen Them  <a href="#">L1</a>, <a href="#">L2</a>, <a href="#">L3</a>, <a href="#">L4</a>, <a href="#">L5</a>, <a href="#">L6</a>, <a href="#">L7</a>, <a href="#">L8</a>, <a href="#">L9</a>, <a href="#">L10</a>, <a href="#">L11</a>, <a href="#">L12</a> </li> <li>Are they real fixes or not?  <a href="#">L1</a>, <a href="#">L2</a>, <a href="#">L3</a>, <a href="#">L4</a>, <a href="#">L5</a>, <a href="#">L6</a>, <a href="#">L7</a>, <a href="#">L8</a>, <a href="#">L9</a>, <a href="#">L10</a>, <a href="#">L11</a>, <a href="#">L12</a> </li> <li>Cross Code Talking-Writing Matrix: Same Sound in Different Spellings  <a href="#">L1</a>, <a href="#">L2</a>, <a href="#">L3</a>, <a href="#">L4</a>, <a href="#">L5</a>, <a href="#">L6</a>, <a href="#">L7</a>, <a href="#">L8</a>, <a href="#">L9</a>, <a href="#">L10</a>, <a href="#">L11</a>, <a href="#">L12</a> </li> <li>Creating Word-Specific Spellings in Mind's Eye  <a href="#">L1</a>, <a href="#">L2</a>, <a href="#">L3</a>, <a href="#">L4</a>, <a href="#">L5</a>, <a href="#">L6</a>, <a href="#">L7</a>, <a href="#">L8</a>, <a href="#">L9</a>, <a href="#">L10</a>, <a href="#">L11</a>, <a href="#">L12</a> </li> <li>Scribes Paying Attention to Letter Order and Letter Position  <a href="#">L1</a>, <a href="#">L2</a>, <a href="#">L3</a>, <a href="#">L4</a>, <a href="#">L5</a>, <a href="#">L6</a>, <a href="#">L7</a>, <a href="#">L8</a>, <a href="#">L9</a>, <a href="#">L10</a>, <a href="#">L11</a>, <a href="#">L12</a> </li> <li>Choosing the Correctly Spelled Word  <a href="#">L1</a>, <a href="#">L2</a>, <a href="#">L3</a>, <a href="#">L4</a>, <a href="#">L5</a>, <a href="#">L6</a>, <a href="#">L7</a>, <a href="#">L8</a>, <a href="#">L9</a>, <a href="#">L10</a>, <a href="#">L11</a>, <a href="#">L12</a> </li> </ul>
<b>Minds In Motion</b>
<ul style="list-style-type: none"> <li>Minds in Motion Introduction  <a href="#">L1</a> </li> <li>Connecting Words through the Musical Melody of Language  <a href="#">L1</a>, <a href="#">L2</a>, <a href="#">L3</a>, <a href="#">L4</a>, <a href="#">L5</a>, <a href="#">L6</a>, <a href="#">L7</a>, <a href="#">L8</a>, <a href="#">L9</a>, <a href="#">L10</a>, <a href="#">L11</a>, <a href="#">L12</a> </li> <li>Creating Meaning By Combining Words  <a href="#">L1</a>, <a href="#">L2</a>, <a href="#">L3</a>, <a href="#">L4</a>, <a href="#">L5</a>, <a href="#">L6</a>, <a href="#">L7</a>, <a href="#">L8</a>, <a href="#">L9</a>, <a href="#">L10</a>, <a href="#">L11</a>, <a href="#">L12</a> </li> <li>Adding Glue Words to Hold the Other Words Together  <a href="#">L1</a>, <a href="#">L2</a>, <a href="#">L3</a>, <a href="#">L4</a>, <a href="#">L5</a>, <a href="#">L6</a>, <a href="#">L7</a>, <a href="#">L8</a>, <a href="#">L9</a>, <a href="#">L10</a>, <a href="#">L11</a>, <a href="#">L12</a> </li> <li>Sentence Word Order  <a href="#">L1</a>, <a href="#">L2</a>, <a href="#">L3</a>, <a href="#">L4</a>, <a href="#">L5</a>, <a href="#">L6</a>, <a href="#">L7</a>, <a href="#">L8</a>, <a href="#">L9</a>, <a href="#">L10</a>, <a href="#">L11</a>, <a href="#">L12</a> </li> <li>Changing Word Order  <a href="#">L1</a>, <a href="#">L2</a>, <a href="#">L3</a>, <a href="#">L4</a>, <a href="#">L5</a>, <a href="#">L6</a>, <a href="#">L7</a>, <a href="#">L8</a>, <a href="#">L9</a>, <a href="#">L10</a>, <a href="#">L11</a>, <a href="#">L12</a> </li> </ul>

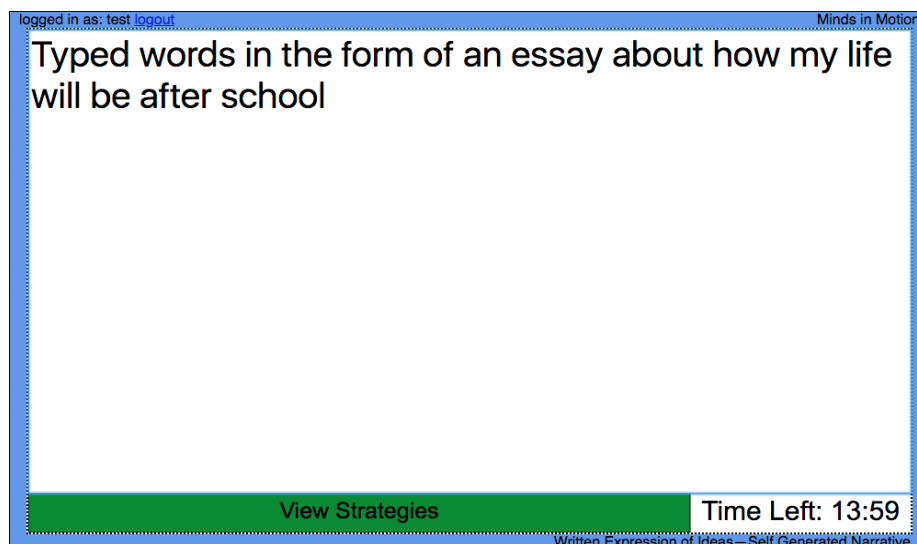
Figure 3.2: Index of lessons from the HAWK 5 version.



Figure 3.3: Screen from the Letters in Motion section of HAWK.



Figure 3.4: Screen from the Words in Motion section of HAWK.



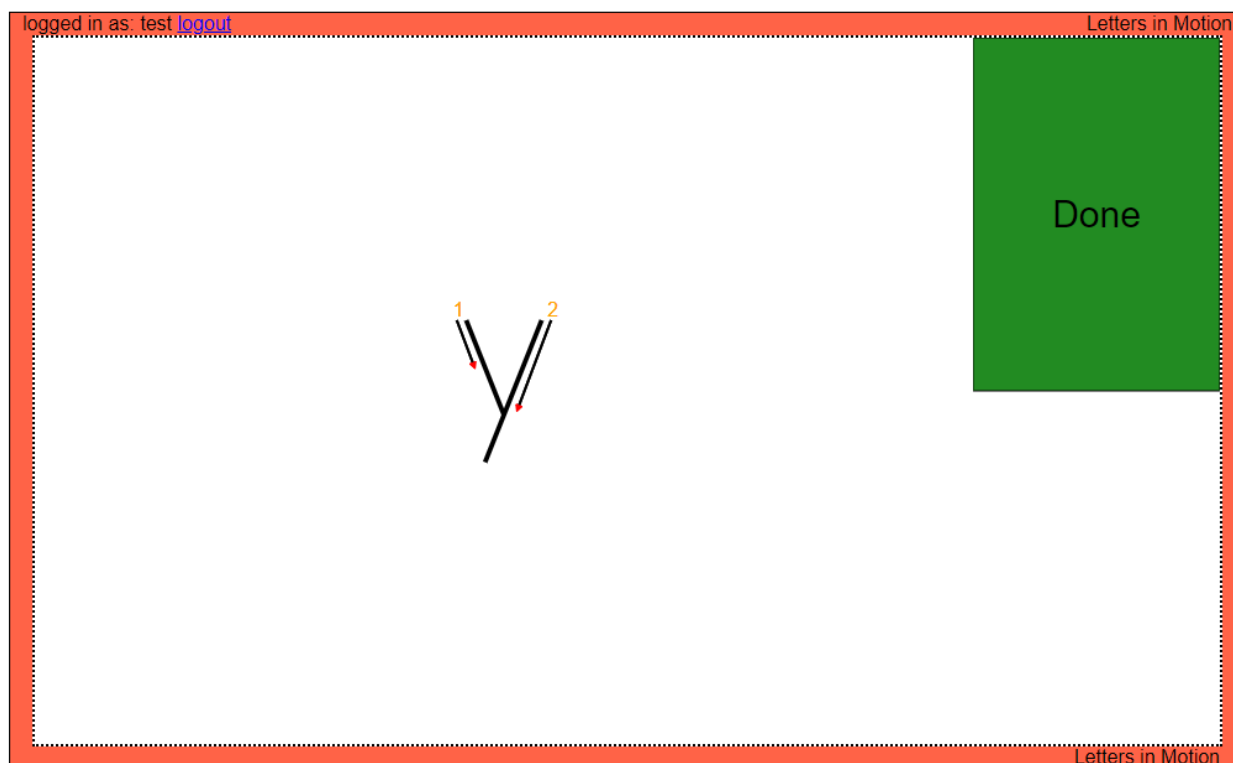
*Figure 3.5: Screen from the Minds in Motion section of HAWK.*

In total, HAWK has hosted more than three dozen distinct activities across its six versions. Many activities are mechanically similar, but all train students on unique concepts or skills. We will describe a representative sample of activities in greater detail to give the reader a sense of the breadth of topics and experiences offered in HAWK.

### **Letters in Motion**

The letters in motion activity is the only one that exists in the letter in motion category, which is why they share the same name. This activity has been present in HAWK from the very first version and has undergone numerous changes since. In the last version tested with students, Version 5, Letters in Motion tasked participants with writing single letters by tracing over example patterns with a stylus. Then the same letter is shown being “drawn” by the computer through animation. Then finally the student is asked to write the letter again unaided. This process repeats for all letters of the alphabet. In different weeks students practice writing upper- and lower-case versions of print and cursive letters in this format.





*Figure 3.6: Screenshot of the Letters in Motion lesson, where students practice writing individual letters with stroke order and direction guides.*

### **Pattern Analyzer Through the Ear**

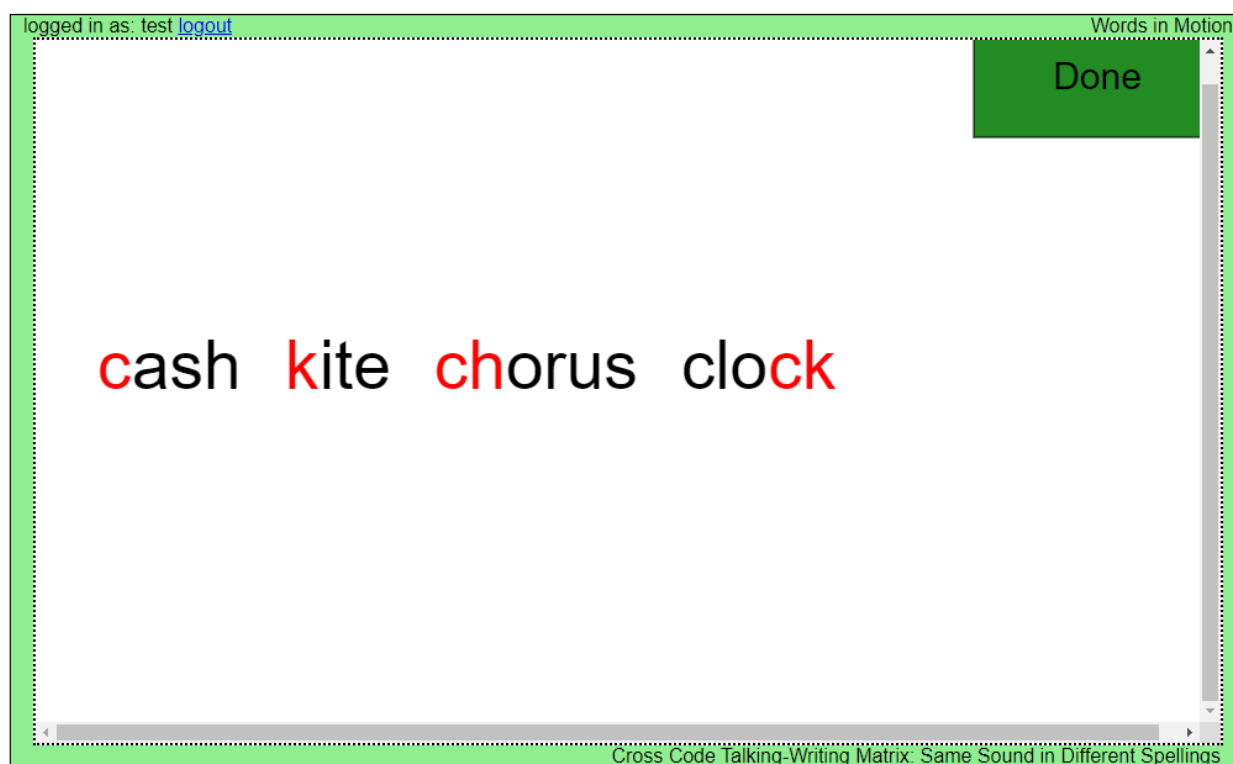
This is a Words in Motion activity where students hear a word said by the computer and then must tap an on-screen drum with the correct number of syllables that occurred in that word. This repeats for several words and then the student is informed of how many words they counted correctly as a score.

### **Combining Two Words to Shorten Words**

This activity in Words in Motion teaches students about contractions. The student is shown two words and then asked to recall and write, with a stylus, the word that results from combining them together. For example, the student hears “you have” and must write “you’ve”. This repeats for several words, with the student being shown the correct answer after every word as feedback.

### Cross Code Talking-Writing Matrix: Same Sound in Different Spellings

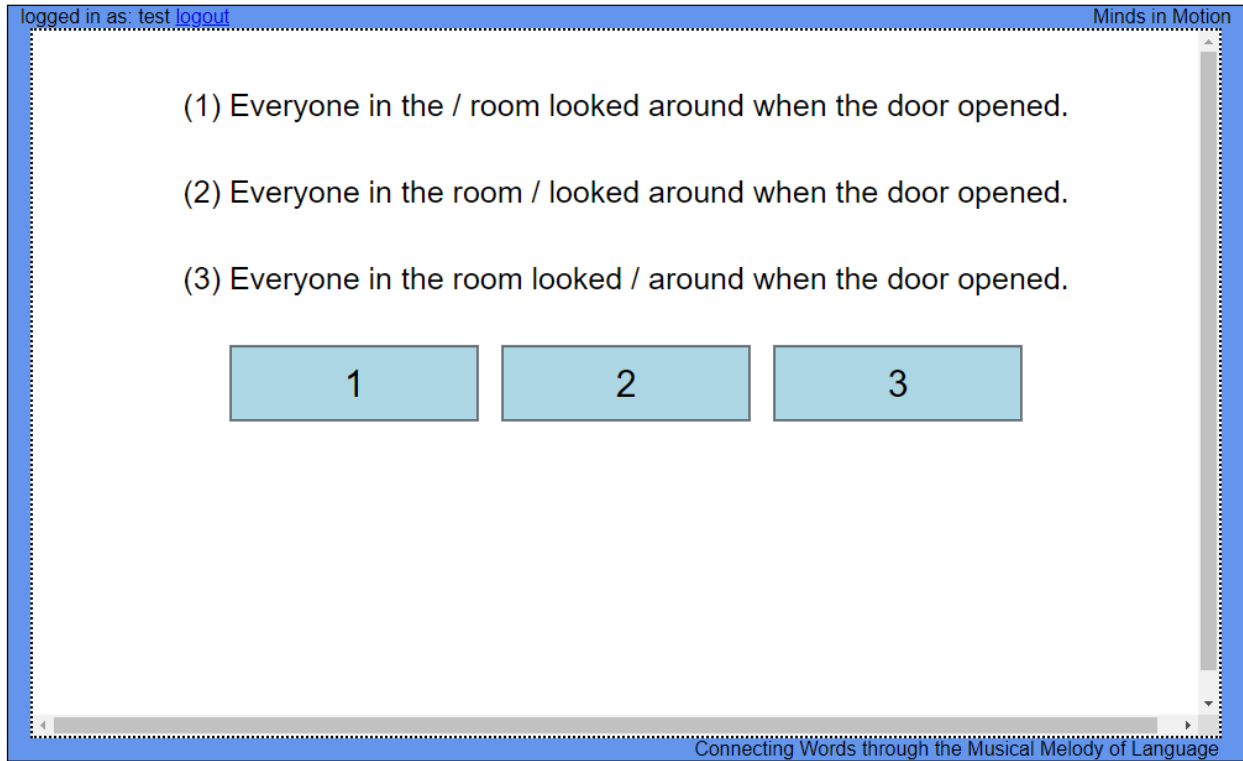
This is a drill activity in Words in Motion where students practice reading, hearing, and saying the different ways one sound can be spelled. For instance, the 'c' in cash, the 'k' in kite, the 'ch' in chorus, and the 'ck' in clock are all pronounced the same way. The student sees a word with the relevant part highlighted in red. Students must try pronouncing the word themselves, then tap it to hear the ideal pronunciation and pronounce it again matching the computer example. This repeats for all of the main sounds in English and their possible spellings.



*Figure 3.7: Screenshot from the Cross Code Talking-Writing Matrix activity where students are drilled on the different ways to spell the same sounds in English.*

### Connecting Words through the Musical Melody of Language

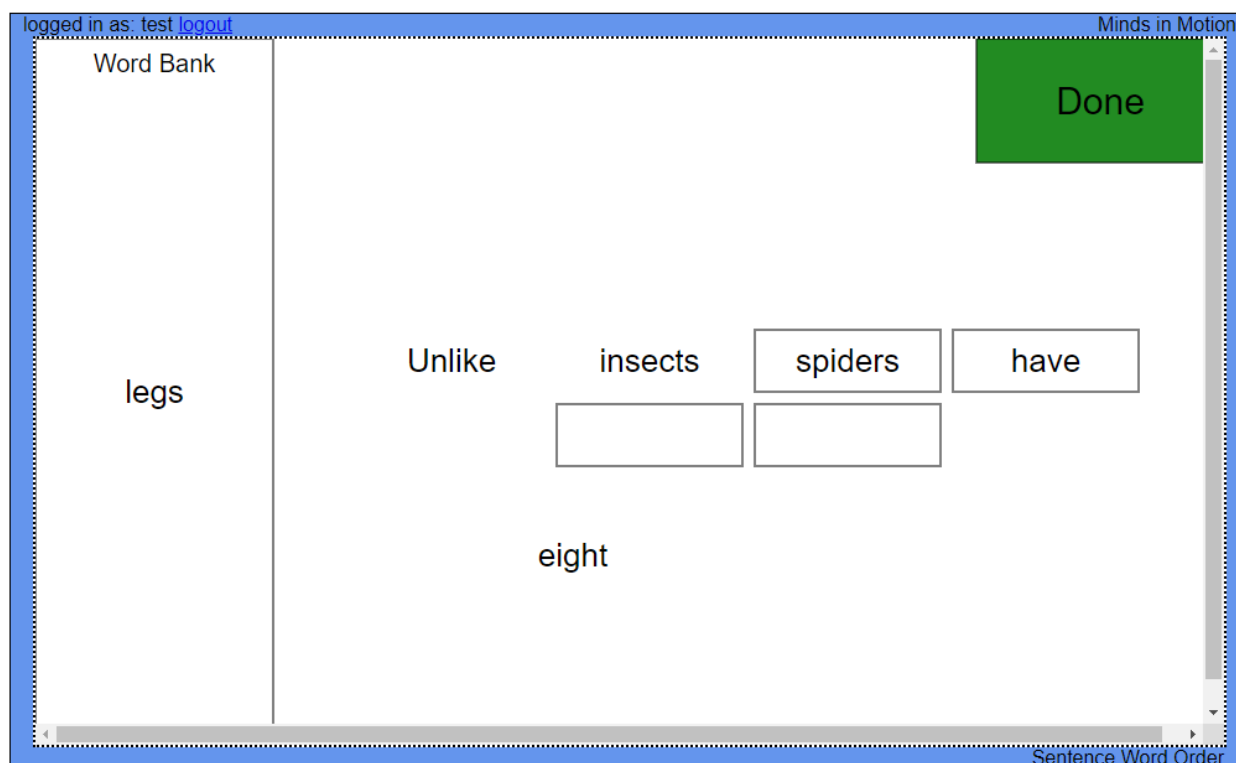
In this Minds in Motion activity students are given a sentence and asked to decide where in the sentence there are natural pauses. They are given several options to choose from and are told if their answer was correct or not afterwards as feedback.



*Figure 3.8: Screenshot of the Connecting Words through the Musical Melody of Language activity, where student pick the example that shows the most natural pause locations in each sentence.*

### **Sentence Word Order**

In this activity, part of Minds in Motion, students are given a set of scrambled words and are asked to click and drag them to rearrange them into a grammatically correct sentence. Students are shown the correct ordering afterwards as feedback.



*Figure 3.9: Screenshot of the Sentence Word Order activity, where students must rearrange words to construct a correct sentence.*

### **Keyboard Warmup**

In this Minds in Motion activity students follow audio instructions to practice finger placement to learn touch typing.

### **Written Expression of Ideas - Integrating Listening and Writing**

This activity is one of several composition activities in Minds in Motion. In this one, students listen to a two-minute lecture about a topic while taking typed notes. After the lecture ends they have ten minutes to type a summary of the lecture using just their notes and memory.

### **Kokopelli's World**

Discussed in much greater detail in the next chapter and throughout this thesis, Kokopelli's World nonetheless represents a distinct type of activity present in HAWK.

Since every student went through all types of activities each week, HAWK was not meant as a system to teach entirely new skills, but rather to practice and re-frame reading and writing techniques in ways that were more appropriate for students with learning disabilities or students who had had incomplete or unsuccessful instruction in the past. It was core to the instructional intent of HAWK to cover all complexity levels of language in the same session. To emphasize this, each new activity was introduced to a student by explaining what they would be doing and also why the activity would be useful for their development. The three sections of HAWK also had their own introductions that contextualized the importance of reading and writing and tried to increase student interest in the topics. Words in Motion, for example, discussed English as a “code” that the software would teach them how to crack and compared it to the vital work Navajo code talkers performed during World War Two. These sections were also used to introduce the importance of storytelling, which would later be carried on into Kokopelli’s World. The Kokopelli figure himself comes from HAWK as well. His original role in HAWK was to dance with joy on-screen every time a student completed a lesson.

The features just described in this section remained static through the course of HAWK’s development, but HAWK went through 6 major iterations that saw changes in the type and number of activities involved. Each iteration involved different research hypotheses as well. The list below gives a broad sense of the focus of each iteration.

**Version 1:** Letter handwriting development through the ductus feedback method (discussed in Appendix A) as well as comparison of stylus-writing vs finger-writing skills.

**Version 2:** This version removed the ductus method writing feedback and instead relied on self-assessment while keeping track of how quickly students completed writing tasks.

**Version 3:** This version introduced typing activities, particularly typed essay portions in the Minds in Motion section. This opened up the opportunity to compare writing performance using handwriting or typing modes.

**Version 4:** Version 4 involved explicit touch-typing practice in contrast to previous versions which assumed some familiarity with touch typing on the keyboard.

**Version 5:** This version incorporated programming materials for the first time (KW) and was focused on exploring how the programming affected the reading and writing instruction and vice versa.

**Version 6:** Also known as HAWK Special. In this iteration HAWK was separated into three versions corresponding to dyslexic, dysgraphic, and OWL LD student groups. Each version contained a subset of activities from HAWK version 5 that would be specifically useful for individuals with that learning disability type. This version was developed but never tested with students.

HAWK Version 5 saw the introduction of the first instance of Kokopelli's World, which will be discussed in greater detail in its own chapter. For the purposes of HAWK however, Kokopelli's World was originally added to HAWK to explore several topics. One, we believed there would be value in teaching programming at the same time as reading and writing. Two, we hypothesized a system that framed programming as a writing task, and supported this in its design, would particularly benefit from simultaneous reading, writing, and programming instruction. We hoped to see the effect in both directions, with programming improving and contextualizing the natural language instruction and vice-versa. Three, to see if the programming would better keep the students engaged through the lessons. Four, there was little to no prior research examining the

effects of dyslexia and dysgraphia on programming instruction or learning, particularly among children.

## 3.2 Results

As mentioned earlier, HAWK was used in several studies and underwent several iterations. In this section we highlight and explain some of the most important findings from HAWK studies. Below that are listed some of the other findings made through one or more studies with HAWK.

Perhaps most importantly, it became clear that even with early iterations of HAWK students using it showed measurable improvement. A portion of the dyslexic student subjects using HAWK showed gains in oral reading (10/17 students) and spelling measures (16/17 students) (Berninger et al. 2015). Both were third party measures that are broadly accepted in the educational assessment community. Importantly, both of these measures are also predictors for dyslexia. In other words, some of the dyslexics using HAWK saw some of their deficiencies lessen. This result was integral to the whole study around HAWK as well, since pre and post-treatment brain scans were looking for changes in brain function. This established HAWK as an effective instruction platform for its primary audience. Though less relevant to this thesis, the same study also saw improvement from dysgraphic and OWL LD students in measures related to their own conditions.

A separate study found that LD students using HAWK improved in measures not linked to their conditions (Tanimoto et al. 2015). This suggests that HAWK may be effective for typically developing students as well as those with learning disabilities.

- Students showed stronger improvement in handwriting, word reading, spelling, and syntax writing when instructed with writing lines and numbered arrow cues for stroke order than without (Tanimoto et al. 2015).

- Students writing with a stylus on iPads did not make a significantly different number of transcription errors than students writing with fitted pencils on paper (Thompson et al. 2016).
- Students wrote more words and with fewer spelling mistakes using a touch typing method than when allowed to type with a “hunt and peck” method (Thompson et al. 2016).
- Students took more notes from written lesson material than from audio lesson material (Thompson et al. 2016).
- Students presented with writing strategies through the computer applied them appropriately when writing summaries of read and heard material (Niedo et al. 2016).
- Coded writing by children showed no significant difference when in note-taking or summary-writing modes, or between writing with a stylus and using a keyboard (Richards et al. 2017).
- fMRI scans of children’s brains showed increased connectivity in certain brain regions during reading comprehension activities after 12 weeks of using HAWK (Richards et al. 2017).
- Dyslexic students using HAWK showed statistically significant improvement across numerous measures of dyslexia (Thompson et al. 2017).
- Dyslexic students demonstrated better reading comprehension with one word shown at a time than having one word added to the total text at a time. (Thompson et al. 2017).
- Coding lessons in HAWK noticeably increased student involvement and motivation for HAWK. Students also preferred coding lessons over the reading and writing lessons (Thompson et al. 2017).



### 3.3 Conclusion

The HAWK system was a key component in a large-scale study on the nature of learning disabilities in children. It led to numerous findings related to writing methods, motivation, as well as coding. Due to this, HAWK also had a strong influence on the design of Kokopelli's World. When it came time to create KW we looked to HAWK as an example of a successful platform for literacy instruction. The next chapter will go into detail on the design features of KW.

## 4. Kokopelli's World Design

Kokopelli's World (KW) is an online programming environment designed to teach typically developing and dyslexic students in the 8-15 age range a set of fundamental coding concepts including looping and conditionals. Through this we explore what specific difficulties learning disabled students encounter while learning to program that set them apart from their typically developing peers, and what a coding environment can do to address these difficulties. This section will discuss the history and design rationale of KW and present a full list of its features. The next chapter discusses the experiments and analyses involving KW to date as well their results.

### 4.1 Kokopelli's World System Components

Kokopelli's World has two main components: the definition and support for a blocks language, and the activities and infrastructure that students interact with. These two components and their respective details are diagrammed in Figure 3.1.

The blocks language of Kokopelli's World is a modified form of Google Blockly (Fraser 2013). Each block has a specified appearance and a type that limits the other blocks it can connect to. It also has a method for generating JavaScript (JS), and logic for when the block is rendered as audio either statically or dynamically. When a student decides to run their blocks program, it is first converted to browser-friendly JavaScript code through blocks conversion methods. This JavaScript code is then run in the JavaScript-based Acorn JS parser (Acorn 2019). The JavaScript commands trigger animations in the program microworld, which are shown in D3.js-based animation and Howler.js-based sound effects and narration (Bostock 2019; Simpson 2019).

Kokopelli's World's lessons are managed through a code library we call the "lesson engine" that includes: the methods to check program state and construction, the representation of the current lesson and knowledge of what lesson will come next, methods for communication with

the server including data recording, and the code for initializing the lesson. Each individual lesson is represented in a static JSON data file that specifies the lesson's available palette of blocks, what checks of the code are required to pass the lesson, the initial program and microworld the user starts with, any hints, and the instructions related to the lesson, including text and audio. When a lesson has begun, the lesson engine loads the appropriate lesson data file, initializes the environment, and from then on responds to user input as necessary to execute or check code and occasionally report usage data to the server.

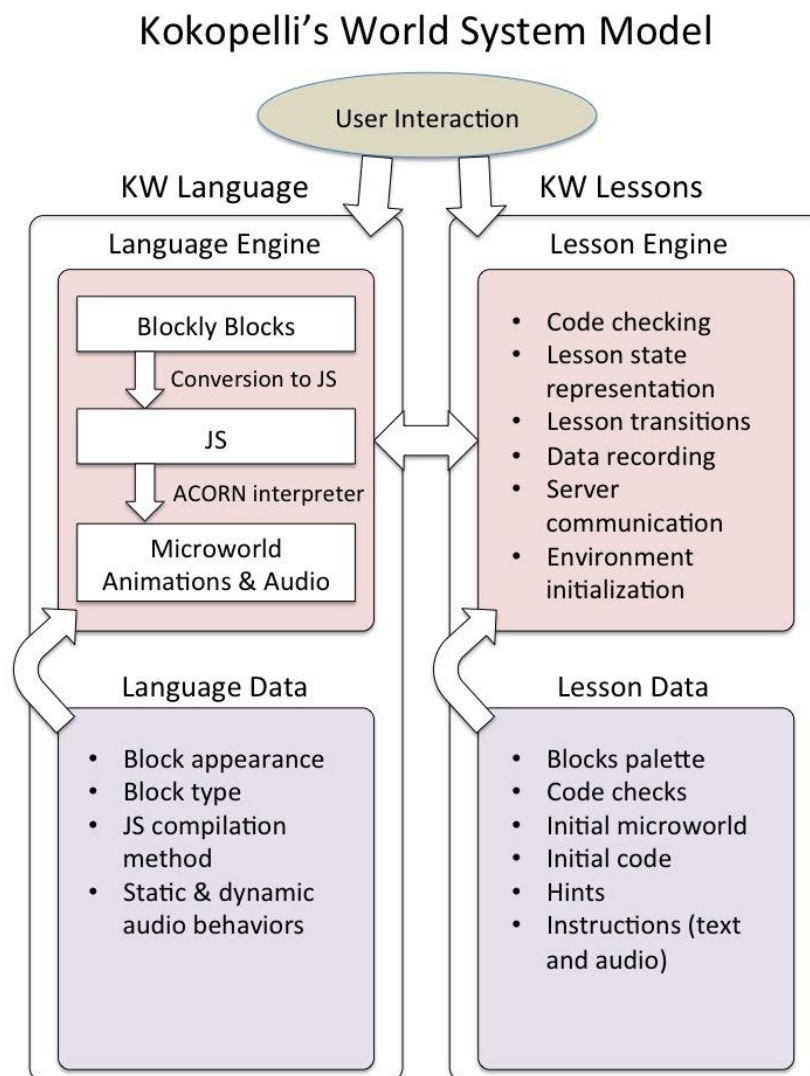


Figure 4.1: A diagram of the core components of Kokopelli's World and how they interact.

## 4.2 Kokopelli's World Features and Design Rationale

As mentioned in the Help Agent for Writing Knowledge (HAWK) section, Kokopelli's World was originally designed to be presented as part of a battery of reading, writing, and coding lessons all targeted at children with dyslexia. This was envisioned as a set of discrete, online lessons that could be done with minimal supervision. Towards this end we structured Kokopelli's World as a series of predefined problems or "puzzles" each with a well-defined goal. These puzzles let us focus the student's attention on a smaller subset of concepts, and also make it easier to automatically assess student progress and learning. Smaller discrete lessons are also more appropriate for strict time limits, which was the context for KW's original development. Each lesson is targeted towards one or more core concepts. Lessons are completed in order so that they can gradually build on the concepts used in previous lessons, such as learning loops before learning about nested loops.

In KW's first iteration as part of the HAWK system, each lesson had a predefined time limit. If a student had not found the solution within the time limit they were allowed to progress to the next lesson anyway. We decided on this because students using HAWK were under strict time constraints and this was the first time students had engaged with the system. This kept the students moving but occasionally resulted in a student progressing without having demonstrated mastery of a concept yet. On subsequent versions of KW we removed this feature and gave students unlimited amounts of time to try each lesson but they were then required to complete the lesson correctly to advance.

To have KW better fit in with the other activities in HAWK, we modeled much of its design around the pre-existing reading and writing lessons. KW commands are all complete English sentences, with capitalization, to make them more familiar and readable to students learning natural English. Figure 4.2 contrasts two commands performing similar actions. The top command is from KW while the bottom command is from the popular Scratch blocks programming environment. The KW version is more descriptive, though as a tradeoff it is less generalized. Reading a KW command

is a natural form of feedback, as students can simply read their code to see if it is a valid command.

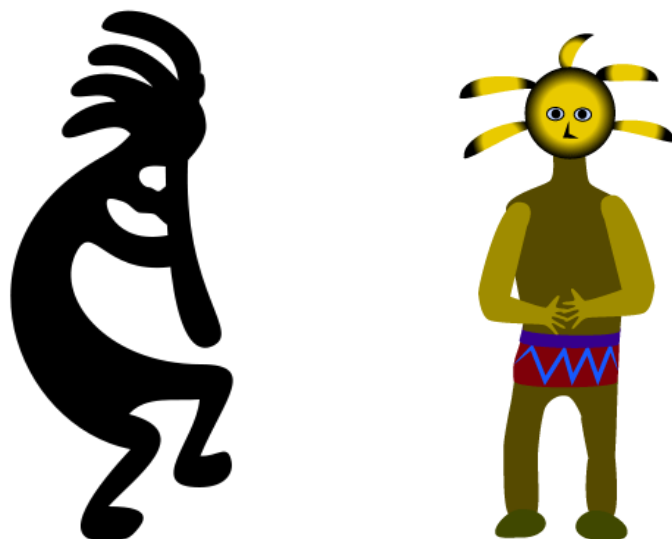
A non-grammatical sentence is also an incomplete command.



*Figure 4.2: Similar commands in KW (top) and Scratch (bottom)*

Commands are also in declarative instead of imperative form. This more closely resembled the language and storytelling style students had been using in the Minds in Motion section of HAWK just before starting KW every week. We suspected it would help transfer skills if the language modes matched. We also speculated that indirect commands in this form might be less frustrating for students when the computer inevitably fails to do exactly what they wanted at some points. It would not mean that the computer was disobeying a (potentially in the student’s mind) direct order that an imperative command implies. Lastly, when the audio output option is invoked, the code execution reads much like a traditional story would. If a command were in regular imperative form, an audio reading of the program would suggest the computer is programming and commanding itself, rather than the user. Figure 4.2 also shows this contrast.

The “main character” of KW is, naturally enough, Kokopelli. Kokopelli is a Native American mythic figure that appears in several cultures originating from the American southwest, including the Hopi, Pueblo, and Hohokam people (Wikipedia 2019). He is typically characterized as a trickster-type god and is often depicted playing a flute. Native American culture and stories appear several times throughout HAWK and Kokopelli is shown dancing at the end of every activity, so it was a natural choice to make him the focus character in our coding lessons as well. Raven, another common character in Native American myth and stories, also appears in some KW activities as a secondary character.



*Figure 4.3: Popular silhouette depiction of Kokopelli (left). KW's more detailed depiction of Kokopelli facing the viewer (right).*

Also as part of the Native American storytelling tradition and the emphasis of natural language in the coding, KW frames programming as a form of storytelling in its activity introductions. The metaphor being that the student is someone creating a story which the computer can then act out through the characters and objects in the microworld. This metaphor is bolstered by the full English commands and declarative language. As discussed in the prior work section, the storytelling metaphor has been used in numerous environments to motivate students and contextualize their work. Coding as storytelling grants more power to stories, as they can be brought to life through execution. This execution also doubles as a form of feedback. If the story does not go as the author intended, then there was a clear mistake in their code.

Kokopelli's World is a blocks-based programming environment. Covered in more detail in the related work section, one of the primary appeals of a blocks-based environment is that syntax errors can be reduced or even eliminated entirely. With fewer errors coming from an often-cryptic compiler, the hope is that students are more likely to stay engaged and not get discouraged by small mistakes. Another useful feature of block environments is they surface all possible commands to the

user through the block palette. Students do not need to read documentation to know what commands are possible because they are all immediately discoverable. Blocks also eliminate the possibility of misspelled keywords that go unrecognized by the interpreter. Kokopelli's World takes this a step further and uses the block palette as a form of hint. Typically, most if not all of the blocks palette for a given lesson will be required for the lesson, limiting the number of superfluous blocks a student could waste time on.

Kokopelli's World includes a microworld, another feature that often appears with blocks environments and for good reason. Microworlds provide a visual space where students can more easily predict and follow program execution, as well as keep track of program state. They are also more engaging for some children than a standard text output. Figure 4.4 shows the KW interface including the microworld on the right. Finally, microworlds were a good pairing with our storytelling theme as the colorful visuals can spark imaginations and animations can match the audio narration provided by the computer.

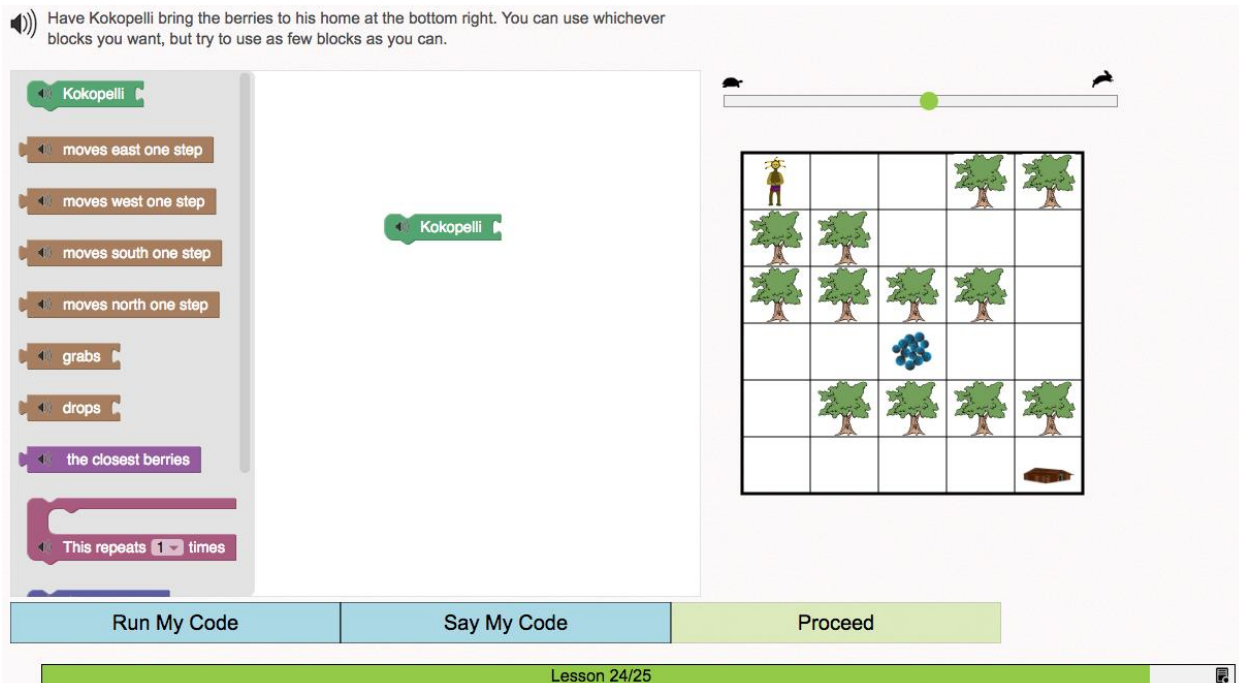


Figure 4.4: Kokopelli's World interface with the microworld visible on the right side.

Kokopelli's World has no variables that are revealed to the student. We made this decision as a way to limit the scope and complexity of concepts students had to tackle at once. As a side benefit, it also made structuring the code as English text simpler. To still allow control structures within this limitation KW has a few workarounds. Loops do not have a visible looping variable. Instead, students must explicitly specify how many times a loop repeats. Conditionals either concretely reference unique objects, such as "Kokopelli" or "drum #2" or use a specially-made block to dynamically reference an object, like "the closest berry", relative to the subject of the command and the state of the microworld at runtime. User input is handled as a conditional block, where each branch is triggered by a different keypress. Since our system is single-threaded, user input is always a blocking action.

To still give students the opportunity to creatively apply the concepts they have learned, we also introduced "open-ended" lessons to KW, though the majority are still pre-defined. These typically involved full access to all the blocks that had been taught up to that point and a generous or unlimited time limit. This gives any students interested in more creative applications the chance to try them out and is the main opportunity for students to flex their storytelling muscles in KW. To prompt the students and prevent "blank page syndrome" each open-ended activity has a unique initial microworld setup.

Open activities were further expanded with projects, a feature that was added in later versions of KW. Projects are also separate sandbox environments that allow the student to use any and all blocks in KW with no assigned goal or time limit. In projects, students can also specify the starting state of the microworld and define their own actor objects that can move around and interact with the microworld in the same way Kokopelli and Raven can.



### 4.3 Kokopelli's World and Dyslexics

Without a strong theoretical basis for which to predict what specific programming concepts, if any, dyslexics would have more difficulty understanding, we chose to use KW to address issues that dyslexics have exhibited in other subjects, and to connect KW to those subjects from HAWK as much as possible. In particular, we focused on reading and spelling difficulties.

Outside of their usual appeal, blocks serve a purpose specific for dyslexics. Dyslexia is commonly known as a reading disorder but can affect handwriting and writing in general. Spelling mistakes, among others, can be used as a metric for dyslexia (Pearson 2009). Along with the above justifications, we chose to use blocks to minimize the amount of spelling dyslexic students would have to do.

A common technological solution to dyslexia-related reading problems is to bypass the issue and convert text to speech. This was also our tactic in KW. Blocks programs can be converted to audio and played to students in several different ways. Blocks also are easier to be read by a computer than text code since it establishes a set vocabulary. In earlier versions of KW we recorded voice lines ourselves. These were changed in later versions to synthetic text-to-speech when it became an option universally supported by internet browsers. This change meant that audio options were still available when students added their own dialog, or custom blocks. Running synthetic text-to-speech on a non-blocks environment could cause problems though. For instance, the computer could correctly pronounce a keyword homophone, confusing the student about what was being done. For example, the computer could pronounce “say” and “sae” the same way even if only one is the recognized command to make a character speak.

Kokopelli's World allows for three different ways of converting code to audio. We call these three options “block reading”, “static reading”, and “execution reading”. These features are unique to KW and are exclusively part of our experimental condition.

*Block reading* is the most low-level of the KW audio options and, when enabled, adds a small speaker icon to all blocks in the environment. At any time, the student can click an icon and the computer will say aloud the text contained in the block. Figure 4.5 shows an example block with the audio icon. This also applies to blocks with changeable text, such as dialog blocks. This option will only read the targeted block and is not affected by any connected blocks. It is meant as a quick way to hear what a block represents if the student is having difficulty reading a single block's word or phrase.

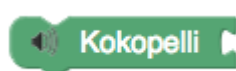


Figure 4.5: A block in KW with the audio output icon visible on the left.

The *static reading* option converts the current block code structure to audio, including all control structures and any code that might not be executed during runtime. For example, if the student triggers the *static reading* option on the program seen in Figure 4.6, the output audio will be “If Kokopelli is near the closest berries, then Kokopelli moves east one step, otherwise Kokopelli moves west one step.” This option was intended as a way to hear an entire program as narration in the case that the student was having trouble reading the code. As the program is read, the block that is currently being mentioned by the narration is highlighted in yellow. This allows the student to follow where the narration is in the code structure.

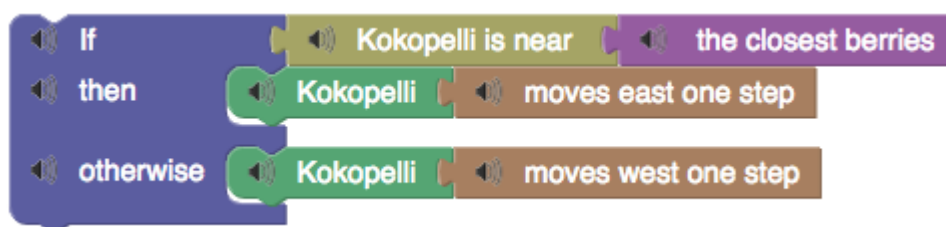


Figure 4.6: A conditional block program that will output different information when read with the *static* or *execution* audio output options.

The *execution reading* function says the program's text out loud as it is being executed in the microworld. Since code is formatted to be grammatically correct English, this amounts to reading code statements as they are executed as well as when control structures branch, such as when a for loop repeats or a conditional branch is chosen. For example, with the same program from Figure 4.6 and assuming Kokopelli is near berries at runtime, the audio output with *execution reading* will be "Kokopelli is near the berries. Kokopelli moves east one step." Note that the other branch of the conditional is not read out loud in this option, and only the result of the conditional is said. In particular, we suspected this option would be useful during debugging for clarifying why the computer entered or exited a control branch. This option is only allowed during code execution, and when paired with the microworld animations, to avoid confusing students.

Outside of the audio options in the experimental condition, every version of KW also includes an audio output format for all instructions and feedback. We have made these available in both the control and experimental versions because they mirror the support that would be available in other blocks environments. Our own informal survey of screen reader support for blocks environments has shown that off-the-shelf screen readers can read static, selectable text but not most block text.

The decision to remove variables from KW was particularly important for dyslexics as they can sometimes demonstrate reduced short-term memory (Berninger 2010). Reducing the number of variables they must keep in their head was another attempt to design KW for this population.

## 4.4 Kokopelli's World Block Descriptions

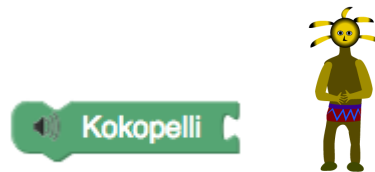
Below are descriptions of the blocks found in Kokopelli's World. Blocks in KW are categorized by their roles in sentence structure. The language has subjects, objects, verbs, and a final "other" category that represents more complex structures, including all of the control blocks.

#### 4.4.1 Subjects

Subject blocks are the root of most commands in KW, and any contiguous groups of commands will either start with or contain a subject block. Aside from control blocks, they are the only other blocks with vertical connection slots to allow commands to stack on each other. Subject blocks determine which character will act, which in turn limits the number of agents to the number of subject blocks available. Unlike some object blocks, subject blocks are never parameterized so there can only be one microworld actor that corresponds to a particular subject block.

To better mimic English it was important for us to structure subjects the way we did. Nearly every command in KW requires a subject, so it was a logical choice to make the subject the root of each command. The pattern <subject> <verb> <object> is also a common and simple sentence structure that we thought would be familiar and appropriate for dyslexic students who may have trouble with more complex grammar structures.

Kokopelli's World has two predefined characters that can each be subjects of commands. Kokopelli is the first character introduced and is present in all activities. Kokopelli's distinguishing ability is he can move heavy rocks.



*Figure 4.7: Kokopelli subject block and corresponding microworld icon.*

Raven is the second character in KW and is introduced later on. Raven's distinguishing ability is she can fly directly to any unobstructed location that is specified by the user.



*Figure 4.8: Raven subject block and corresponding microworld icon.*

#### 4.4.2 Objects

In addition to subjects, Kokopelli's World has several object types. These are objects in an English grammar sense of the word, but they are also objects that exist in the microworld that characters and other objects can interact with. The primary distinction between subjects and objects is that subjects can enact verbs, while objects cannot. Subjects also have matching object blocks to let them be targets of some verbs. In designing KW it was important for us to offer multiple objects and forms of interaction with them. An environment that had too limited a scope would bore the students quickly. As an environment with a focus on storytelling, it was also important to create enough objects to allow for a variety of narratives.

Berries are objects that can be picked up, carried, dropped, or eaten by characters. They do not obstruct characters' movement. Berries can be moved and eaten to create friendly interactions between characters and also show that actions can have permanent effects on the microworld state.



*Figure 4.9: Berries object block and corresponding microworld icon.*

Rocks are objects that can be picked up and carried by Kokopelli but cannot be eaten. They also obstruct character movement. Rocks present more complex navigation problems than in

lessons prior to their introduction and Kokopelli and Raven have to deal with them in different ways, either by moving them or flying over them respectively.



*Figure 4.10: Rock block and corresponding microworld icon.*

Trees cannot be directly interacted with, obstruct movement, and generally act as an immovable barrier. Trees were a practical necessity and act as interior walls in the microworld space while still being thematically appropriate.



*Figure 4.11: Tree icon. Trees do not have an associated object block.*

Drums can be hit by a character to cause a sound to play. They come in seven different sizes, which play different pitches when hit. Drums were added to highlight KW's audio options along with the reading affordances, and different drum pitches mean the user can create simple music in KW.



*Figure 4.12: Drum block and corresponding microworld icon.*

The home, or “goal”, is another object, like the tree, that does not have a corresponding block. It is used for specific lessons to mark where a character should end up after the program execution is finished. The home object was added as a way to more organically refer to a particular space than just a coordinate position.



*Figure 4.13: The Home icon modeled after a Native American longhouse. Home does not have a corresponding object block.*

#### 4.4.3 Verbs

As with the subject blocks, verb block choices were largely informed by the types and depth of stories we wanted to enable in KW. A verb block connects to a subject block and may also include a connection for an object block.

“Move” is the most basic and common verb available. Characters can move one space per command in any of the four cardinal directions. There is also a “moves randomly” block that randomly picks one of the four directions each time it is run. Movement is an integral action in any microworld, as otherwise characters can only act at a distance. Character location is also a state property that is intuitive to even programming novices.

“Picks up” will cause a character to pick up and hold an object, assuming the object is in the same space or an adjacent one. Only one object can be held at a time. The related “drop” action will cause a character to put down whatever object is being carried, with the exception of the rock, which will be put down in the cell in front of the character. The pick up and drop verbs encourage more subjective interpretations such as giving, taking, and trading items and also mechanical

complexity when students must keep track of what has been grabbed and when inanimate objects must be moved to a new location.

“Eat” will remove a food object from the microworld, assuming it is in the same cell as the subject character or in an adjacent space. Eating is another story-rich action and also alters the program state.

“Dance” will cause the subject character to perform a dance animation but has no effect on the microworld. It is a simple action that doesn’t require an object block but adds interest to the actors. It has shown to be a popular block with students.

“Say” is one of few blocks that accept keyboard input. The say command will cause a character to speak whatever is typed into the block during program execution. The text-to-speech is pitched down during dialog to differentiate it from the audio narration used in the three experimental audio output options. The Say block was an important inclusion in KW to support storytelling. While it does not alter the program state, it does let students give voice to their character(s) and apply their own preferred personalities and motivations. No lesson requires specific text to be in a Say block, and it supports text-to-speech like other blocks so we felt this rare instance of writing in the KW system would still be appropriate for dyslexics.

“Hit” will cause the character to hit an adjacent object. This action fails when the actor tries to hit another character and has no other response unless a character hits a drum. If a drum is hit, a tone will play whose pitch is determined by the size of the drum.

“Fly” is a verb that only works with Raven. With Fly, Raven can move directly to a specified location without having to navigate around obstacles. The destination location must still be unobstructed and must be explicitly specified as an (x,y) coordinate before runtime, meaning users must first understand coordinates to use it properly.

“Turn” is a verb that will change a character’s orientation by 90 degrees. This changes the character’s appearance and is useful for deciding where Kokopelli will drop a rock, for instance. Turn is an advanced action for carefully manipulating rocks.



“Appear” will cause new objects or characters to appear at a specific location in the microworld. If the object is unique and already in the microworld, such as a character, it will be moved to that location instead. Until projects were introduced in KW Camp this was the only way a student could add objects to the microworld. The corresponding “Disappear” block will directly remove a character or object from the microworld.

#### 4.4.4 Other Blocks

The conditional block has two branches for one or more commands as well as a slot for a Boolean conditional. Conditional blocks can be nested with themselves and other control blocks.

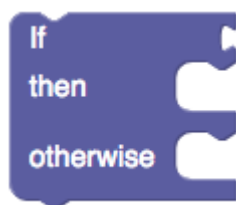


Figure 4.14: The conditional block.

There are two types of conditions available in KW. The first is the “you say ‘yes’” condition that prompts the user with a yes/no dialog box at runtime. The student’s response directly determines which conditional branch is taken. The second condition, “is near”, is a binary operator that takes two objects and returns true if the objects are in the same or adjacent squares. Is near in particular required us to create object copies of subject blocks since so many situations involve checking if a subject is near an object. We determined them to be necessary to maintain the <subject> <verb> <object> construct throughout KW. The block typing system ensures that students will not be able to connect the wrong block to their program when picking between subject and object versions.



*Figure 4.15: The two condition blocks testing user input (top) and relative location of two objects (bottom).*

The loop block holds any number of commands and repeats them a student-specified number of times, up to five. These blocks can be nested with themselves and with other control blocks. Five iterations was a somewhat arbitrary limit we imposed based on the average size of most microworlds in KW. Since loop blocks can be easily nested to create longer loops we deemed this limitation acceptable. Our choice to eliminate variables also encouraged this.



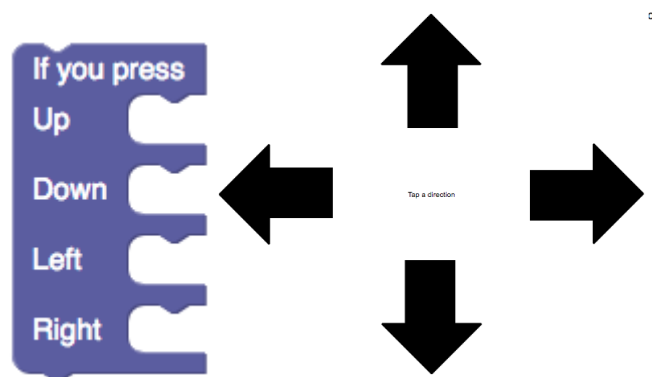
*Figure 4.16: The loop block.*

The Forever loop block operates just like the loop block above but has no halting condition. Since KW has no break statement, this means programs running this block will have to be stopped by hand and no commands placed after this block will ever execute. Forever blocks are used primarily in KW when user input and game-playing is introduced. Outside of an infinite game loop we have not seen this block used very often but it is very appropriate for that particular instance.



*Figure 4.17: The Forever loop block.*

The direction block is a four-branch case statement meant to facilitate player input. When the block is executed, an overlay will appear on screen with 4 directions and prompt the player to click or tap one. Whichever direction is chosen, the corresponding branch will execute. Kokopelli's World was originally used with iPads which prohibited us from relying on keypresses most of the time. Instead we settled on a custom UI overlay whose four buttons map to four case statements. While these could be used to create any sort of four-way control branch, we intended this as another construct for use with game-making. We expected most students to use this block to directly control a microworld character's movement.



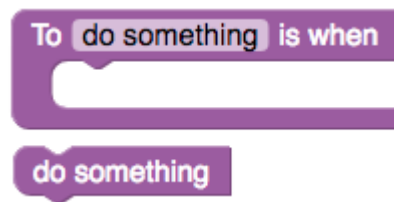
*Figure 4.18: The direction block (left) that accepts user input and the UI overlay (right) that appears when the direction block is executed.*

The “The end” block serves no programmatic purpose, but instead is used to support the metaphor of the KW programs as stories. It is the only statement block without a bottom connector so that it always marks the end of a contiguous set of blocks. Most lesson sets in the KW HAWK version ask students to finish their programs with this block to remind them about the storytelling concept and to mark the end of the day's work in HAWK.



*Figure 4.19: The end block.*

Functions in KW are student-created and student-named. Upon being defined, a function statement block is automatically generated that can be used to call the function. Any number of functions can be made by the student and recursion is allowed. Functions mark the upper limit of what we expected students to do in KW. The code editing window in KW is fairly small but the total space is infinite, so functions become necessary to abstractly represent code that is off-screen. Function names are the other instances where typed input is accepted in KW. Like with the Says block, the content of the typed text does not matter so we were not concerned with issues of spelling that can occur with dyslexic students.



*Figure 4.20: A student-defined function definition block (top) and a block calling that function (bottom).*

## 4.5 List of KW Lesson Types

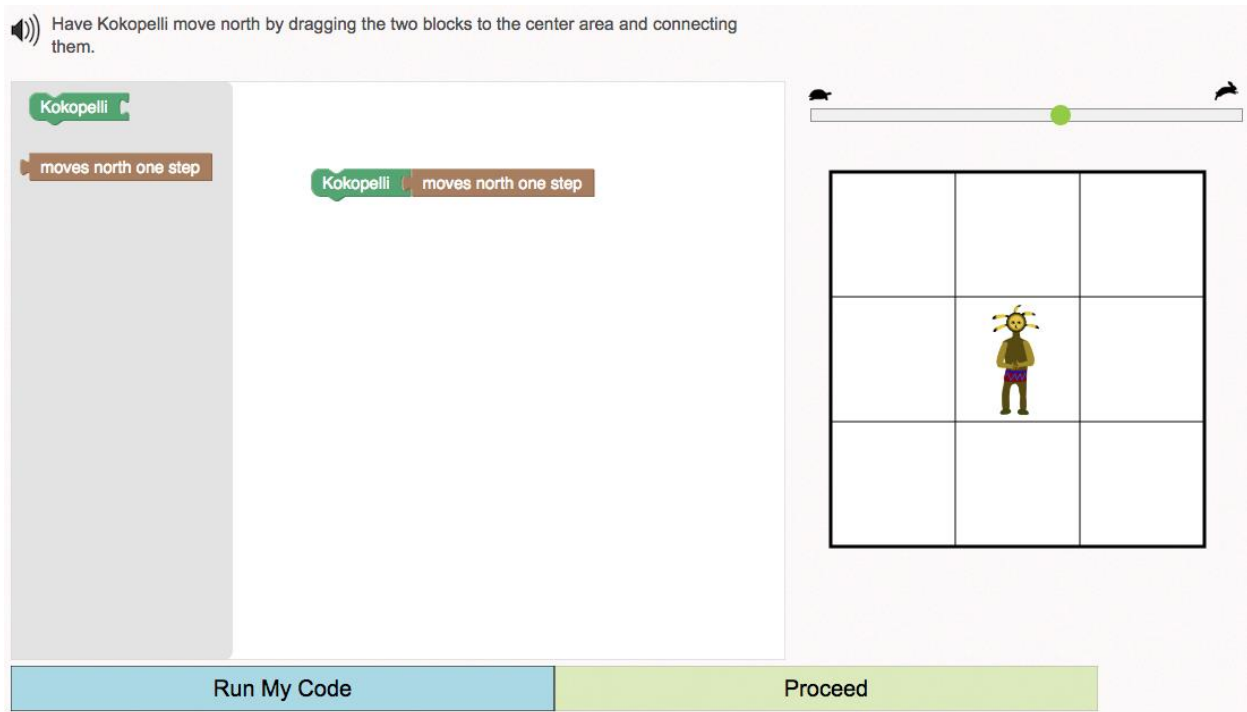
### 4.5.1 Introductory lesson

The introductory lesson does not involve any actual programming on the part of the user. This lesson instead introduces the student to the KW interface by highlighting important areas before they must create any blocks themselves. Simply running an empty program is enough for a student to complete this lesson.

### 4.5.2 Lessons introducing system concepts

These lessons are meant to teach students the basic fundamentals of blocks programming and to gradually familiarize them with the non-control-structure commands they will have to use in later,

more complex, lessons. Students will learn how to create blocks, how to delete them, how to connect them together into valid commands, how to vertically stack commands, and how to use the move, grab, eat, dance, and drop verbs. HAWK users also learned the coordinate system, to specify individual cells with the Appear and Fly verbs.



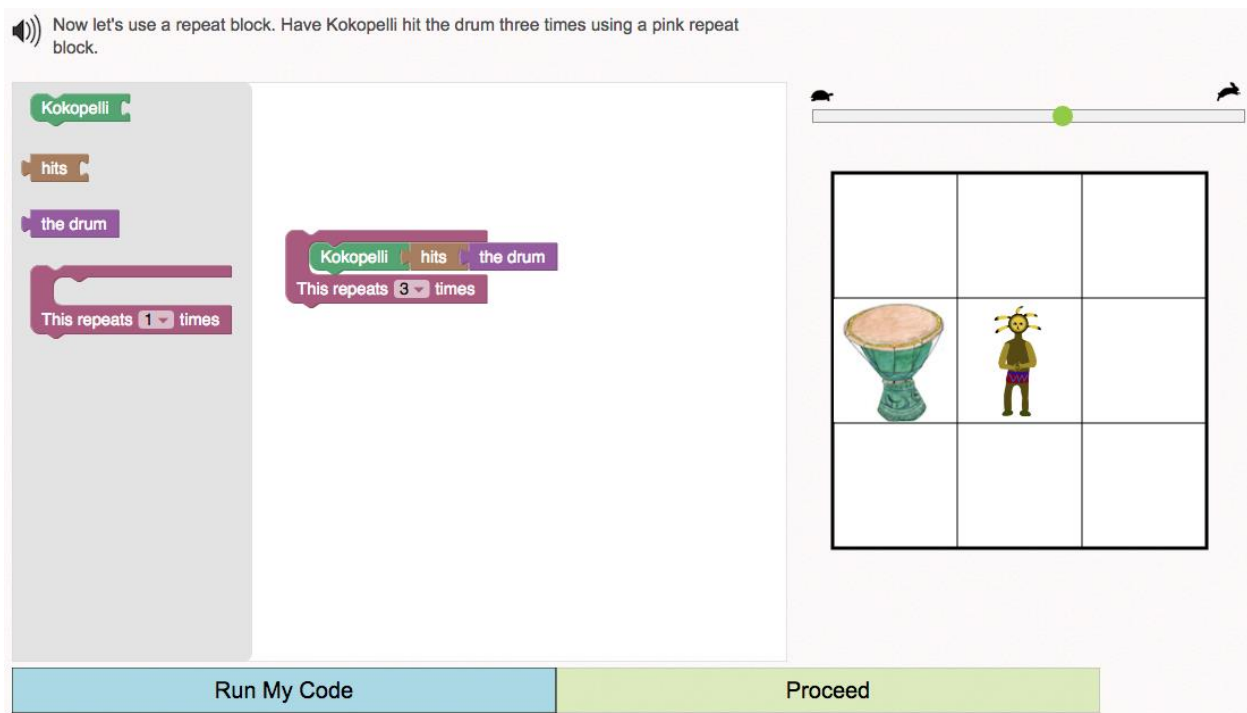
*Figure 4.21: Activity in KW teaching the student how to connect blocks horizontally to create a command.*

#### 4.5.3 Lessons with loops

These lessons mark the students' first experience with control structures, loops in this case. Students will learn how to create loops and specify their number of iterations. They will learn how to stack commands both inside of and around loops, and how to use multiple loops in sequence. The end of this set of lessons marks the halfway point in KW.

We decided to introduce loops before conditionals for several reasons. For one, we reasoned loops would be simpler than conditionals for the students to understand. Loops only have

one slot for blocks to be added, rather than three with conditionals. Loop control flow is also static and controlled directly by the student before runtime rather than dynamically determined during runtime by the microworld state as with conditionals. Secondly, loops allow the student to reduce redundant block use. At the point loops are introduced students are already moving characters large distances across the microworld one step at a time. We deliberately created these situations to better communicate the utility of repetition when it finally became available.



*Figure 4.22: Activity in KW introducing the students to the loop block by having them hit a drum three times.*

#### 4.5.4 Lessons with Conditionals

Conditionals are first introduced to students at the start of the second half of the lessons. Students will first see how conditionals are used, and how they can branch depending on the current program state, before trying it themselves. They will learn about the two conditional types available in KW: the condition that asks for a yes/no choice from the user and the conditional that checks if two objects are within one space of each other. Conditionals based on object proximity are common

tests for early conditionals across blocks languages. For example, the Hour of Code tasked programmers with directing a character through a simple maze using conditionals to check if a character was near a wall or not (Code.org 2014). Students will also learn how to use multiple non-nested conditionals in the same program by stacking them on top of each other.

Conditionals can be a difficult programming concept for beginners since they are often the first instance of a branch in control flow. They are also a tricky concept to ground in the metaphor of programming as storytelling. Without a way to change the initial microworld a student may see a program as deterministic and thus it can be unclear why a conditional is even needed when behavior can be pre-specified. To ease students into the idea we took two approaches in different versions of KW. The HAWK version of KW introduces conditions with the “If you say ‘Yes’” block. This condition halts execution until user input is received so it very clearly marks when the condition check happens. The binary response, yes or no, will also be familiar to even very young students. Since conditionals are introduced before randomness, and students cannot edit the microworld starting state, user input is the only non-deterministic factor in KW available at this point in the progression. It is difficult to convey the importance of conditionals with a simple, non-deterministic program, particularly since we wanted to avoid using nested control structures until later lessons.

In later versions of KW, we added a new introductory lesson for conditionals that used a pre-made program with a conditional. Students must run the program once, see how it executes, and then change the microworld state and run the program again to cause a different conditional branch to be taken. We made this change to force students to watch the same program execute twice under different conditions and to give them more practice after students using KW with HAWK showed they still did not have a strong grasp of conditionals.

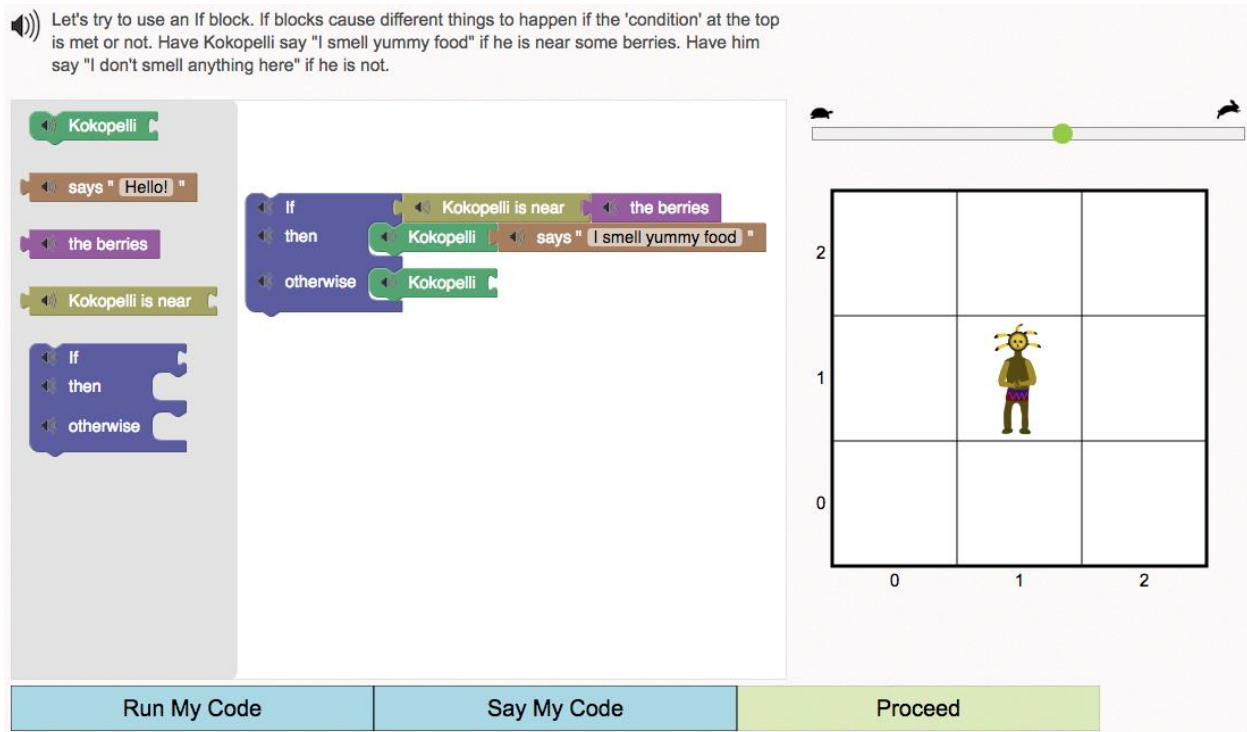


Figure 4.23: Activity in KW letting users try if blocks for the first time. In this case, Kokopelli says different things if he is near the berries or not.

#### 4.5.5 Lessons about Nested Control Structures

The final set of lessons in the current iteration of KW involves learning how to nest loops and conditionals to create more complex programs. The end-goal of this set of activities is to create a program that causes Kokopelli to systematically explore the full microworld and eat any berries he encounters without any failed actions, e.g. running into a wall or trying to eat berries that are too far away. These lessons create a complex task in a small microworld space to make students appreciate the power of nested control structures. Without control structures, a program performing the same task would be very long, repetitive, and likely involve many failed commands



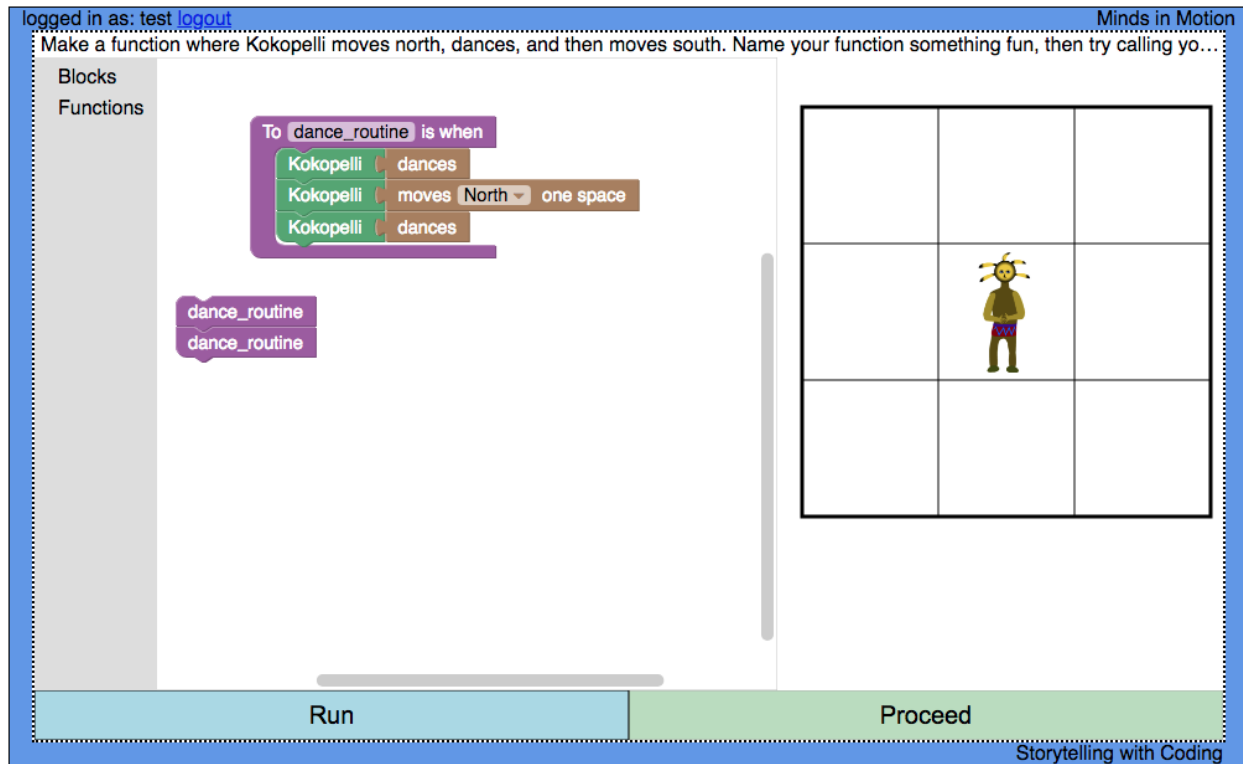
Have Kokopelli move east three times with a repeat block. Each time after he moves, an If block should check if you say yes. If you do, he eats the nearest berry.

The image shows a programming interface with a script area on the left and a game area on the right. The script area contains a 'Kokopelli' block, followed by 'moves east one step', 'says "Hello!"', 'eats', 'the closest berries', an 'If' block with 'then' and 'otherwise' branches, 'you say "Yes"', and a 'This repeats 1 times' block. A nested loop structure is shown with 'Kokopelli moves east one step', an 'If' block with 'you say "Yes"', and 'This repeats 3 times'. The right side shows a 4x4 grid with a character and berries.

Figure 4.24: This KW activity introduces nested control structures to create more advanced behavior. In this case, Kokopelli moves east in a loop and only eats berries when given permission by the end-user.

#### 4.5.6 Lessons about Functions

KW supports the option for parameter-less functions. Students can create, name, and invoke any number of parameter-less functions they wish. Recursion is possible in KW but not explicitly taught.



*Figure 4.25: KW HAWK activity teaching functions. Students are asked to create a dance routine for Kokopelli as a function that can be called multiple times.*

#### 4.5.7 Lessons about Randomness & End-User Input

The final lessons included in KW HAWK deal with teaching the students the beginnings of randomness, with a new block that causes a character to move in a random direction. They also learn about more detailed forms of end-user input, with a block that will take in a user-specified direction and call one of four conditional branches as a result. These are the last elements we determined would be necessary to make simple games in KW, which is often of interest to child programmers. Research has shown that males in particular are more often motivated to create and play games (Kelleher 2007). Since the majority of dyslexic individuals are male (Miles 1998), we deemed it prudent to include these elements. To maintain the metaphor of storytelling, we frame these programs as “interactive” stories where the end-user and computer work together.

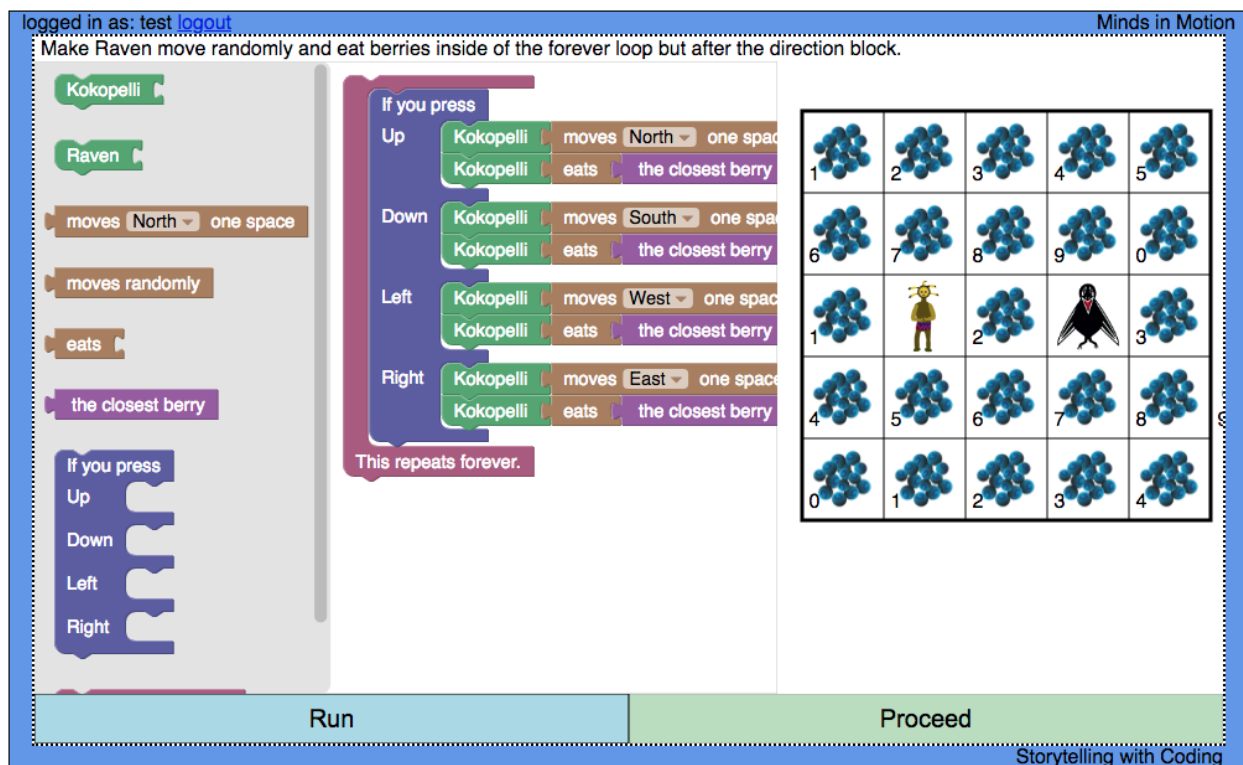


Figure 4.26: The final KW HAWK lesson that incorporates end-user input and randomness to make a rudimentary game where the player competes with the computer to eat more berries.

## 4.6 Differences Across KW Versions

There have been three versions of KW to date. These versions are: the original design that was present in HAWK (KW HAWK), a standalone version adapted for a summer coding camp (KW Camp), and another standalone version adapted for use in classrooms (KW Class). The experiments and methodology associated with these versions are discussed in the next chapter while this section will focus purely on the functional differences between each version. Those differences are compiled into Table 4.1 below. Any feature not mentioned can be assumed to be present across all versions.

*Table 4.1: A table specifying the feature differences between the three versions of KW.*

	KW HAWK	KW Camp	KW Class
Output program as plain text option	x		
Time-limited activities with automatic progression	x		
User-defined functions	x		
Randomness	x	x	
Open-ended activities	x	x	
Projects		x	
User-defined characters		x	
Synthesized voice text-to-speech		x	x
Pre-recorded human voice text-to-speech	x		
Coordinate system	x	x	
Playback-complete data collection		x	x
System measured student affect		x	x
Characters can face multiple directions	x		
Reading Test		x	x

The option to output programs as plain text was a feature we prototyped in the HAWK version of KW. Students could choose to see their program written in unformatted text while it was

being executed. This was to further encourage the storytelling metaphor. However, since our primary audience was dyslexics, it felt inappropriate to give them more text to read. This was reinforced when we saw it was not a feature that was used often during the first experiment. More complex programs were also not well-represented in plain text, as the feature did not use indentation or parentheses. The content of the text matched what is read as part of the static reading audio option available in KW Camp and KW Class.

The time limitation during lessons was removed in later versions. That decision is explained earlier in this chapter.

Student-defined functions were dropped in later versions of KW to decrease the scope of topics covered. When KW was shifted to an online experiment model with KW Class, we wanted to limit the time frame of KW to at most a couple hours and we thought functions would push the curriculum over that time. Randomness was treated similarly, though it was included in the projects from KW Camp to give the students a larger palette of blocks to use. Neither is introduced in the KW Camp curricula but we decided the random block would be self-explanatory enough to warrant its inclusion. This time constraint is also the reason why the coordinate system, and the turn verb were left out of versions after KW HAWK.

Projects were a feature we wanted to include since the beginning of KW, but there was not enough development time until before KW Camp. They were also particularly appropriate for that setting as the camp was largely about open-ended group project programming. We also added the option for students to create their own characters and use them in place of Kokopelli. The hope was this would make KW projects more appealing to users with prior programming experience who might quickly get bored with KW's default options.

As discussed earlier in this chapter. We switched KW code reading from pre-recorded voice to synthesized voice to allow the computer to read elements with student input, such as lines of dialog.

An implementation oversight in the first version of KW meant that information was missing from recorded events where two blocks were connected together. This meant that it was not possible to reconstruct the student's exact program except at times when it was run and a full snapshot of the program was taken. This error was corrected in subsequent versions of KW and is explained in more detail in the next chapter.

KW HAWK's experiment was run with a researcher in the room who could observe and give paper questionnaires measuring affect. In later versions this format changed and it became necessary to measure student affect as part of the software itself. KW Class and KW Camp ask the student to rate each lesson on a scale of one to five stars to gauge enjoyment.

## 5. Experiments, Analysis and Results

This chapter presents the data format that Kokopelli's World uses, the experiments that have been run with Kokopelli's World to date, and a discussion of the results from those experiments.

Kokopelli's World has so far been involved in two experiments: the study as part of HAWK (KW HAWK) and a study that used KW in a classroom setting (KW Class). Outside of those studies, the KW Camp version was also used informally in a summer camp setting. Lessons learned from that experience are discussed in the next chapter.

To briefly summarize results from these two studies, work with KW has shown that dyslexic students can complete and understand programming activities, and that it can keep those students engaged through several hours of programming activities. Dyslexic students using KW and HAWK largely preferred programming over reading and writing activities. Dyslexic students have demonstrated a set of consistent, recognizable programming patterns that can be used to distinguish and characterize their understanding of programming. In the quantitative space, a reading measure associated with dyslexia has been shown to significantly correlate with coding activity performance. However, another different reading measure associated with dyslexia has shown to not significantly correlate with time spent solving coding activities. Further research is recommended to better explore the relationship between these measures and programming learning. The rest of this chapter will present these results and others in more context and detail.

### 5.1 Data Format

Data collected by Kokopelli's World is broken up into two categories: score data and event data.

Score data is a high-level summary of a student's performance within each lesson. One score is recorded each time a lesson is completed that includes how long the user took to finish the lesson and a value representing the student's affect from the lesson.

Event data is more low-level and is generated each time the user performs an action in the software. Events include the event type, a general data field, the lesson in which the event occurred, and a timestamp. These events are batched together on the client side in groups of up to 50 before being sent to the data server or sent when a lesson ends. An event is recorded every time the user either creates a block, deletes a block, picks up a block, drops a block, connects two blocks together, disconnects two blocks, executes their program, triggers a timeout, or clicks any button. With the complete set of event data we can re-create a timeline of a user's individual actions and identify behaviors that would not be apparent from the score data alone.

These data formats were originally created for the HAWK system. The motivation was to have a standardized form that would be valid across a wide variety of activities from writing data, to keyboard events, to data from multiple choice quizzes. Scores were intended to be data that would require almost no processing and could immediately be used as input for statistical analysis. Event data was intended to be more detailed but require more processing effort before it could be analyzed. When KW was introduced into HAWK it was a natural choice to use the same data format, and that was not changed for subsequent versions of KW.

From the raw event data we identified several measures in which we saw potential for differentiation among students. The most straightforward was time taken to finish each lesson. Specifically, we mean the period of time between when a student first starts a given lesson and when that student exits the lesson to move on. We refer to this measure in shorthand as “time taken”. This measure is used with the assumption that students who complete a lesson quicker than others demonstrate a stronger understanding of the lesson material. This was made more complex in KW HAWK with its automatic progression after a certain time limit, but the time taken to complete each lesson was used, in some form, for much of the data analyses related to KW.

In addition to time taken per lesson we were particularly interested in patterns of block use in KW. Towards that end we counted the occurrences of types of block manipulation for each lesson. Among these “block events” we looked for instances of a block being deleted, being created,



being connected to another block, and being disconnected from another block. The assumption was a student with an abnormally high number of one or more of these types of events in a given lesson might be struggling during that lesson. For example, a student who continually makes and discards various blocks in an attempt to reach a goal, or a student that keeps the same set of blocks but shuffles them around in various configurations to get a desired outcome. Conversely, a student performing a minimal number of block actions during a lesson theoretically demonstrates clear intention and, potentially, a correct understanding of the problem.

## 5.2 Experiment 1: KW HAWK

Kokopelli's World was first used as part of HAWK version 5. This was the first use of KW with actual students, though it was not the first time HAWK had been used. One of our hypotheses was that dyslexic students would be able to learn programming through KW, though potentially with some difficulties. We also planned to look for correlations among student performance in the coding portions of the software, severity of student learning disability, and student reading/writing performance. The last we were particularly interested in due to our framing coding as a writing activity in KW. Finally, we were curious about the qualitative experiences the dyslexic children had with programming in KW, including how much they enjoyed it and how they conceptualized programming after using it.

This experiment involved 15 children in grades 4-6 all diagnosed with dyslexia, 6 female and 9 male. The children came to the University of Washington campus once per week for 12 weeks on weekday afternoons during the summer. In each session the students would go through one lesson of HAWK including the Letters in Motion, Words in Motion, Minds in Motion, and Kokopelli's World sections. These sessions took an average about 1.5-2 hours each for a total of 36-48 hours with the software. Roughly 20 minutes of each of these sessions were devoted to interaction with KW in particular. In addition to the data automatically collected through using

HAWK, students were also given questionnaires about KW and HAWK on weeks 9, 10, and 11.

These questionnaires are shown in full in Appendix D, but in general were centered around determining student background, affect and preference.

Twelve students completed the first questionnaire, while seven completed the second. Most students (8 out of 12) had had some experience programming before using KW, the most common environment being Scratch. Prior programming experience was not quantitatively measured as part of the pretest for this study, but through observations we can say that no student was thoroughly skilled in programming before starting KW.

We asked students if they preferred guided programming exercises with defined solutions or to tell stories in more open-ended, sandbox-style creative tasks. Five out of twelve students preferred the guided programming tasks, five preferred to tell stories, and two showed no preference. On a 1 to 5-point Likert scale, students on average preferred programming activities (4.6 avg.) vs. writing (3.4 avg.) or reading activities (3.9 avg.).

We also sought to find out students' affinities towards specific types of blocks in KW. Among the three control blocks (for loop, conditional, and function), students preferred using the loop block (5 out of 7). Most explained because using it saved them time or it was "interesting." It is likely this is also because they had more exposure to loop blocks than the other two. A breakdown of control block usage statistics in activities where they weren't required, shown in Table 5.1, corroborated these stated preferences. Loops and function block instances were considered "incorrect" if they did not have any contents, or if the function was not called at least once. Conditional blocks were considered incorrectly used if they did not have a complete conditional and code in at least one branch. Students used loop blocks far more often and with greater proficiency than function or conditional blocks.

*Table 5.1: Aggregate correct/incorrect usage statistics for control blocks in activities where they were not required.*

Construct	Number correct	Number incorrect	Total
Loop	23	10	33
Function	3	5	8
Conditional	1	0	1

As part of the lesson sequence, students did two activities that we made deliberately similar. In the “sentence anagram” lesson included in HAWK students had to create complete English sentences by dragging scattered words together in order. In one instance of this activity we had the students create a sentence that was also possible to create in KW: “Kokopelli eats the berries”. During the same session we asked students to create a representation of the same sentence in a KW activity. Afterwards we asked students about their thought processes while solving these two activities. We found that students used similar strategies in the “writing” and “coding” versions of the activity. They knew how to order the words by relying on their existing knowledge of English and created an order that “made sense” when read. This activity may have been too simple to strain the students’ knowledge of either code or writing, but it suggested that, for simple examples, students can think in these two modes in similar ways.

For a quantitative measure of a student’s successes in the coding lessons, we created two scores: “coding success” and “coding fluency”. Coding success was a per-student, per-lesson binary measure indicating if that student had attempted and succeeded at completing a particular lesson within its time limit. Students who did were marked with a one, students who did not were marked with a zero. Students who did not attempt the activity at all did not generate a record.

The second measure was coding fluency. This was an attempt to measure how skillfully a student completed a given activity. Coding fluency we defined as coding success weighted by the

ratio of time taken to complete the activity compared to the fastest student. Thus the fluency for the student who completed an activity the fastest would be one while the other scores would asymptotically approach zero as the time taken increased. Table 5.2 shows the total coding success and average coding fluency for each student. Lessons attempted counts the number of lessons each student participated in at all. Lessons completed is the sum of coding success counts for each student; the number of lessons each student solved within each lesson's respective time limit.

*Table 5.2: Coding success and fluency per student participant.*

Student ID	Lessons completed (coding success)	Lessons attempted	Avg. Coding fluency
164	21	25	0.55
165	48	57	0.57
166	38	52	0.34
167	16	16	0.59
169	16	16	0.57
170	54	61	0.61
171	52	62	0.48
172	51	53	0.66
173	32	40	0.58
174	43	50	0.61
175	48	55	0.58
176	39	46	0.51
177	28	32	0.59
178	27	34	0.56
180	16	26	0.37

In a multiple regression, coding fluency was predicted by two different reading comprehension scores ( $F(2,8) = 5.33, p=.03$ ). These two scores were “alphabet 15” ( $r=.71, p=.01$ ) and “word choice” ( $r=.75, p=.008$ ). “Alphabet 15” was the pretest measure for how many legible letters a student could write in alphabetic order in 15 seconds. “Word choice” was a pretest measure for how long a student took to identify the correct spelling of a word from a list of homophones. Both of these measures are predictors of written language compositional fluency, and word choice is a predictor for dyslexia. Only word choice predicted coding success among the pretest measures

( $r=.845$   $p=.001$ ). Coding fluency also predicted coding success ( $r=.63$ ,  $p=.03$ ). A multiple regression with coding fluency and “one word at a time”, a measure of reading comprehension, was also significant at predicting coding success ( $F(2,9)=5.98$ ,  $p=.02$ ) with coding fluency contributing uniquely ( $t=3.42$ ,  $p=.008$ ). This suggests a link between the reading comprehension methods and coding success through coding fluency.

### 5.3 Experiment 2: KW Class

KW Class was an experiment to see if we could recreate some of the coding-specific results of KW HAWK without including the lengthy reading and writing instruction of HAWK. We also anticipated richer data collection, as we refined our event data capturing process to include more granular steps. This was also to be a larger-scale study than KW HAWK that included typically developing students as well as dyslexics. We knew collecting data from enough dyslexic students to achieve statistical significance would be unlikely. Instead, to place students involved with the study on a linear scale we had all students take a short reading test (explained in the next section). Our main hypotheses for this project was that reading ability would predict some of the coding measures we developed. With the introduction of the KW audio support features to typically-developing students we also expected to see a smaller but measurable effect on student performance based on their use of the audio features in KW. Finally, we anticipated a richer set of student experiences with the more granular data collection.

The KW Class experiment took place over the course of a school year in four classes all taught by the same teacher. The school was a private elementary school in the Pacific Northwest. Two groups were 3rd grade classes and two were 4th grade classes. Students and their parents were invited to participate in the study and participation was not mandatory. Roughly half of the students in each class participated. Sessions with Kokopelli’s World occurred in the late morning or early afternoon for 30-45 minute periods. Students spent all classroom time on KW until they finished all

lessons, which typically took two to five periods total. The classroom teacher and a researcher were present to provide help if students had significant difficulty completing the lessons. In total, 57 students participated in the study, 30 in the 3rd grade and 27 in the 4th. Of these, 31 self-identified as male and 25 as female. Students ranged between eight and ten years old. Three participants had been diagnosed with dyslexia by a professional prior to the study, with several others disclosing other learning disabilities. Being part of a technology class, all students had been exposed to programming prior to the study, though most had had no more than one hour of prior coding experience in the classroom.

Participants in the study went through 25 lessons in KW. All students took the first 14 lessons in the same format. Starting in lesson 15 participants were randomly split into two groups: the control and experimental groups. The control group continued to use the same interface as in the previous lessons, while the experimental group were able to use the three experimental audio features described in the previous chapter.

Attached to this study we also recruited two more dyslexic students to go through the KW materials in a more private setting with just a researcher and the student's guardian. These two students were recruited through personal contacts and were both 11 years of age, one male and one female. These students were not from the same school or class as the other students in KW Class, but did experience the same instructional material. The students answered interview questions about their general opinion of programming before using KW, and their experience with it afterwards. Both students participated in one single session each, lasting 2-2.5 hours.

### *5.3.1 Reading Test*

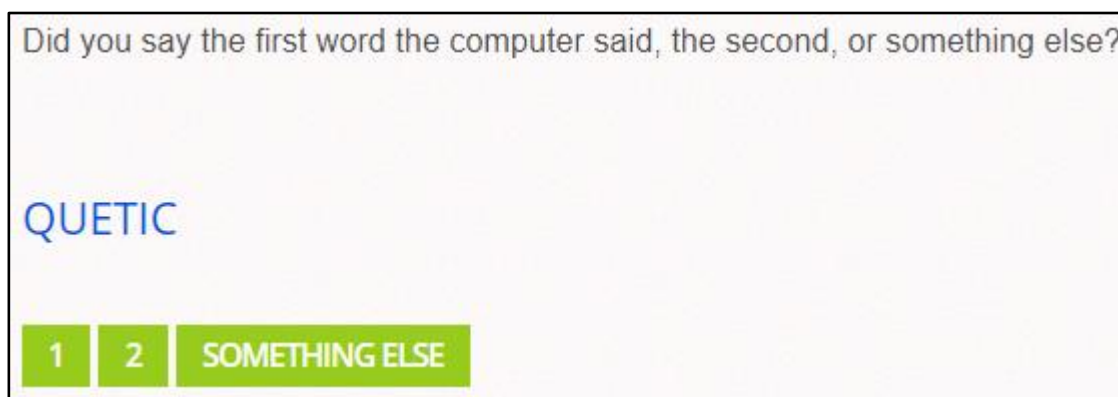
Students involved in the KW HAWK experiment were assessed for learning difficulties as part of a larger research project. This included a battery of pre and post-tests using commercially available proprietary materials applied by trained proctors. With KW Class we did not have the resources to apply the same pre and post test battery. To offset this we chose a lesson from HAWK that

correlated with dyslexia, and closely resembles a test used in the Woodcock Johnson test for Cognitive Abilities, a popular assessment battery that can be used as part of a dyslexia diagnosis (Woodcock et al. 2001). We administered this test for all students before they took the KW coding lessons. Also we collected demographic information from parents. We hoped to see consistency between test performance and dyslexia ground truth data, and to see correlation with test performance and success in the coding lessons. While it is oversimplifying, for brevity we will refer to this test as the “reading test” for the rest of this section.

The reading test as it occurred in HAWK consisted of a list of words, all of which were not real English words, but were constructed with phonemes and syllables such that they seemed like viable English words. During the test, one of these words was shown on the computer screen at a time to the student, the student was prompted to read the word out loud, they were then verbally read the “ideal” way to pronounce the word, and finally asked if they pronounced the word the same way. Students could answer this question “yes” or “no.” After answering, the computer would go on to the next word. This lesson would repeat over the 12-18 weeks of HAWK instruction, with word complexity increasing every 6 weeks from simple, short words to longer, more complex ones.

In the process of converting this test to work in KW we were somewhat concerned about the “honor system” expectation of the test; that students were expected to answer honestly when it would be potentially clear to them what the “right” answer actually was after the computer revealed it. This format was fine under the supervision that occurred during KW HAWK but would be less reliable in the chaotic classroom environment of KW Class. To test if this worry was founded, we split the reading test into two parts. The first part performed as described and as occurred in HAWK. In the second half, we presented words in the same way but students had to choose from three options. Students were asked to choose between two possible pronunciations offered by the computer as well as a third “other” option if their pronunciation did not match either example. Also, unlike the first half, the “correct” pronunciation was not consistently the first option, but randomly distributed between all three. We surmised that students would have a much harder time “cheating”

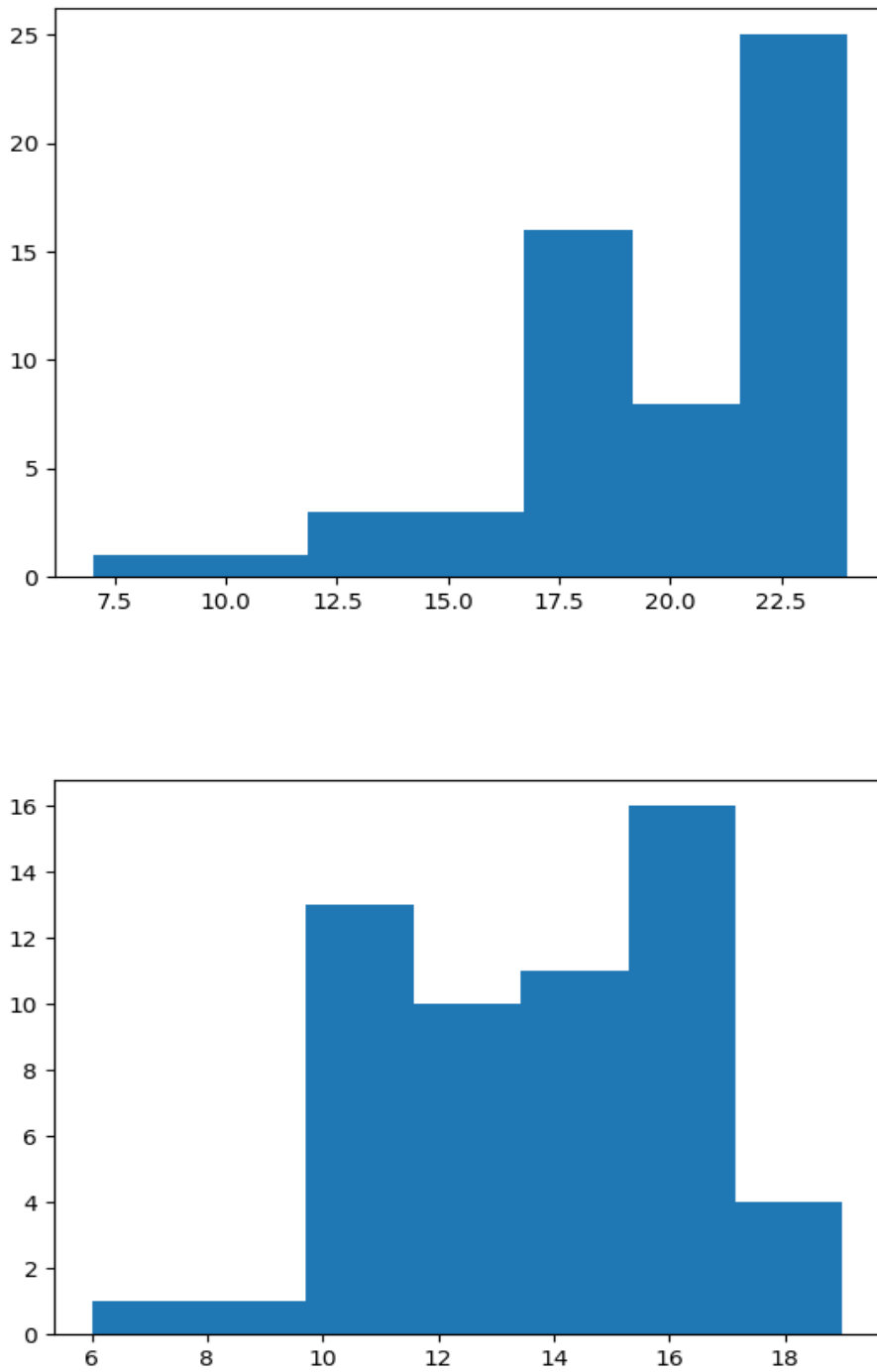
during this form of the test than the original. A screenshot of a single word from this test is shown in Figure 5.1. To make the two halves comparable we distributed an equal number of words to each half and ensured that the difficulty of the words within each test ramped up at the same rate. In the final version, for each half, students would encounter 8 short words, 8 medium words, and 8 long words. All students participating in the KW Class study took the test that resembled the HAWK version before taking the newer one. Words were not repeated across the two halves but were of similar length and complexity and all words came from the original list of words used in the HAWK lesson.



*Figure 5.1: A screenshot of a single question from the second half of the KW reading test. The fake word is shown in the middle with the three possible answers below.*

For the complete test, the possible scores ranged from 0 to 48. Student scores were between 22 and 43, mean 33.8, median 34, standard deviation of 5.2. For part one, several students received the maximum possible score, and on average the difference between the scores in part one and part two was 6.2 in favor of part one. No student received full marks for part two. This supports our concern that some students may not have answered truthfully for part one. Figure 5.2 shows histograms of the two halves. Part one does not appear normal while part two does.





*Figure 5.2: Histograms of the student reading scores from part one (top) and part two (bottom) of the test. Part one does not appear to be from a normal distribution while part two does.*

The Pearson correlation coefficient between part one and two scores was  $p=0.27$ , suggesting only a low correlation between a student's performance on part one versus part two of the test. Considering the close similarity in the actual difficulty of the two parts, and that, if anything, any learning effect should make part two's average score higher than part one, we attribute this to students answering less honestly in part one. Because of these results, only the part two scores from the reading tests were used in subsequent analyses. Any mention of "reading scores" beyond this point are in reference to part two of the reading test.

### *5.3.2 Time Taken*

We had originally planned to use the time each student took to complete a lesson as an important measure of their programming ability, much like in KW HAWK. Our experimental condition complicated things however. Students had some limited control over the speed of their program execution. This control differed between the experimental and control conditions, since the speed of the narration in the experimental condition needed to be slow enough to understand. On average, we expected that a program executed under the control condition would be faster than the same program executed under the experimental condition. Since students could still manipulate blocks during program execution, it also would not be sufficient to just strip out all execution time from both conditions. Instead we normalized total lesson time by subtracting all gaps in time between when a student ran his or her program and the next action performed. If, for example, the student sat and watched the whole program play out over a whole 12 second execution time then 12 seconds was subtracted from the total time taken to finish the lesson. If instead the student started editing blocks only 2 seconds into the program execution then only 2 seconds were subtracted from the total. This, we felt, was the best option to eliminate the effect of the naturally longer execution time in the experimental condition while still allowing that some students were performing critical actions while their programs were running. Figure 5.3 shows the boxplots for the time students took to complete each lesson both before and after normalization. Lesson number is shown on the X axis

and time taken to complete the lesson in seconds on the Y axis. Outliers are represented as circles.

This process reduced the average total time of each lesson by one to three minutes.

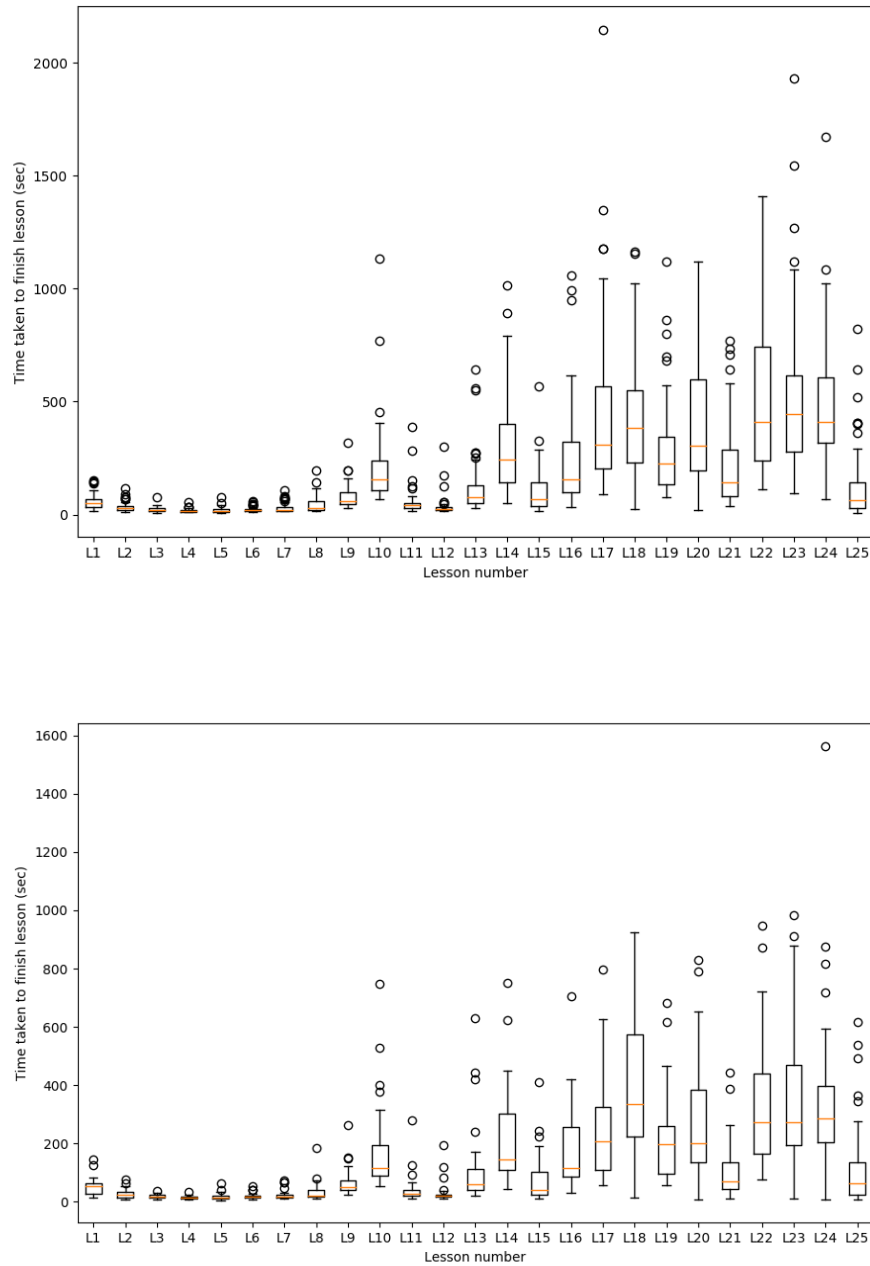


Figure 5.3: Boxplots for the time students took in seconds (Y axis) to complete each lesson (X axis)

both before (top) and after normalization (bottom). Outliers are shown as circles.

Counter to our expectations however, ANOVA tests examining the time taken between the experimental and control groups in the second half lessons did not show any significant time difference, neither in the raw nor adjusted time datasets. The three dyslexic students participating in the study did all take longer to complete all lessons (5847 seconds, 4832 seconds, and 7095 seconds) than the average completion time (4772 seconds) but two of the three students still fall within one standard deviation (1396 seconds) of the average. The full results tables are shared in Appendix C.

An ANOVA test comparing affect scores across conditions showed a significant difference only in lesson 24 ( $F=5.96$ ,  $p=0.02$ ) with the control group reporting a higher average affect. This lesson asked students to navigate Kokopelli through a maze, using whichever coding method they wish. The maze was large enough that many movement commands were required and the repeated narration in the experimental condition may have annoyed students to the point of explaining the difference in affect.

The experimental condition did not appear to have a significant effect on time taken, but students felt a negative difference in one of the longest lessons, and although differences were not significant, average affect was larger for the control condition on almost every lesson. ANOVA tests further revealed that the experimental condition did not have any significant differences from the control condition for any lesson with regards to number of blocks created, deleted, connected, or disconnected.

A visual inspection of time taken per lesson revealed that in most lessons a small group of students took noticeably more time than the average, usually several standard deviations above the mean. This group did not seem to be related to grade, gender, age, or disability. We suspect these were instances where students got distracted during the study. These outliers were not consistently from the same students.

In addition to the experimental condition, we were primarily interested in whether subject reading score had any strong effect on experimental outcomes. We calculated the Pearson

correlation coefficient between reading score and the above measures but did not see any coefficients with absolute values higher than 0.43. A multiple ordinary least squares linear regression over reading score, user condition, and grade - the variables we expected to have the strongest impact on performance - did not predict normalized time taken well for any lesson. No model achieved an r-squared value higher than 0.21.

While it is not possible to prove the null hypothesis, in this case we clearly cannot disprove it. With this value of  $n$  size we can say that reading score does not seem to have a significant effect on performance in KW. Many of the time taken measures demonstrated large standard deviations, which likely muddled the analysis. With larger numbers of students we may be able to detect smaller differences than was possible with  $n = 57$ . These results suggest that reading score for this particular test does not have a strong enough effect on a student's ability to code to outweigh individual differences.

### *5.3.3 Qualitative Results*

We change focus to qualitative results now. First we have the results obtained from the KW Class questionnaire that came at the end of the experiment. After examining that data we also present a more detailed look at the experience several dyslexic students had while participating in KW.

#### **Survey Responses**

As part of KW Class, all students were asked to complete a questionnaire about their experience in KW at the end of the study. Altogether 47 students completed the survey at least in part. Questions were posed and answered in a Likert format on a scale from 0 to 4, or as short response.

When asked how much they enjoyed the lessons on a Likert scale of 0-4, the average response was 2.8. This was higher than the average affect measured after every lesson with the exception of the last lesson (avg 3.3 on the same scale). This suggests students viewed their overall experience more positively than the experience offered by each individual lesson.

When asked how useful the sound was in the lessons the average response was 2.3 on a scale of 0-4. There was no significant difference in averages between the experimental and control condition groups. The audio feature additions to the experimental group do not appear to have affected the students' appreciation of the audio as a whole.

When asked how Kokopelli's World compared to other kinds of programming, 4 (9%) students chose that it was "much worse", 6 (13%) chose "a little worse", 20 (43%) chose "about the same", 8 (17%) chose "a little better", and 8 (17%) chose "much better."

Some 23 (49%) students said they would like to do more lessons in KW if given the chance. This accounts for all the students that answered the previous question positively, as well as nearly half that answered neutrally.

When asked about their favorite block to use, the block most mentioned by students was the repeat block (n=15). By their responses, most students thought it was simple to use and powerful. Some of these responses may have been influenced by the instructional materials, since the introduction to loops explains the use of them in language that reappeared in some students' answers.

The next most popular blocks were the dances block (n=7), the if block (n=5) and the Kokopelli block (n=4). Students mentioning dances mostly conveyed their enjoyment seeing a character dance. Responses mentioning the if block largely referenced its power. One student who answered the Kokopelli block seemed to recognize it as being the root of most commands, and thus important to all other parts of the program. It is encouraging that both control blocks were in the top three mentioned. This suggests at least some students recognized the power of control structures and did not see KW as simply a way to make deterministic, animated stories. Responses also included the says (n=3), hit (n=1), grab (n=1), drum (n=1), and Raven (n=1) blocks.

Students were asked to estimate how many times they used the audio features for an average lesson. This question seemed to have been worded too vaguely as the answers were quite varied. The most common responses ranged between 1 and 5 times in each lesson, though some

students said they did not use it at all and one student answered “a million”. Both the control and experimental conditions included audio instructions and feedback hints. The audio instructions were played automatically for every lesson, though they could be skipped while the feedback hints were entirely optional.

A reminder that the key differences with the experimental condition were the introduction of three new audio features. First was the execution reading that happened automatically when any program was run. Second was the block reading that allowed students to click a block and hear that individual block read. Third was the static reading feature that allowed students to hear program code structure read to them. Only the execution reading was mandatory and played whenever the code was run. The 29 students in the experimental condition used block reading 220 times total over all lessons, averaging out to 7.6 times per student. The static reading feature was used 91 times in total, averaging out to 3.1 usages per student. These numbers are too low to account for the student responses, and the execution reading does not explain the similar responses from the control group. The question was apparently too ambiguous and interpreted differently by the students, but through responses it is clear that most of them felt the presence of the audio features in the lessons.

We asked students to recall their thoughts on programming before starting Kokopelli’s World. Fourteen answered in largely negative terms using words like “nervous”, “confused”, or just “meh”. Twenty-nine students answered positively using words such as “good”, “fine”, “great”, and “happy”. Several students mentioned positive feelings about programming while still feeling some apprehension such as “it was ok sometimes a little hard to understand” and “I felt like there were somethings I was confused”, and “not as confident”. The students in KW Class participated through their technology class, and all had had some exposure to simple programming before, such as Hour of Code, so it does not come as a surprise that the majority of students had a positive outlook of programming even before starting KW.

We then asked students their thoughts on programming after completing Kokopelli’s world. Thirty seven answered positively using phrases like “better”, “happier”, “good”, and “I like it

more”. One student answered negatively and said “confused”. Six students said they did not feel largely different after using Kokopelli’s World. All but one of the students who originally came in with negative feelings on programming left KW with a more positive outlook. Naturally these sorts of questions are subject to positive bias but taken as-is this suggests KW may have a positive effect on students’ perceptions of programming as a subject.

The last set of questions involved students comparing the concept of programming to other activities. We first asked students if they felt like programming resembled being told a story by the computer. Seventeen students said it did not citing various reasons, such as “there was no end” or “because no bad guy”. Nineteen students responded in the affirmative with a variety of explanations including “it had sound like an auto story”, “when raven and kokopelli were together”, and “there was a lot of different scenes”. The rest of the responses were neutral or blank.

Responses here were nearly evenly divided. For students that did not think coding felt like being told a story, one of the common themes that appeared was narrative. Without an antagonist, as one student mentioned, or an explicit end as another did, then perhaps the individual lessons felt too disconnected to function as a complete story. The affirmative responses either recognized the potential for a complete story to exist in just one lesson, like “when raven and kokopelli were together” or were able to derive some sort of meaning from multiple lessons. One student thought that the narrative format was enough to establish a similarity.

We then asked students if they felt using KW resembled telling a story rather than passively being told one. Eighteen students answered yes and 15 answered no. Among the yes responses were reasons such as “he moved when you told him to”, “because it felt like I was animating [a story]”, “because I got to choose what the characters did”. Among the no reasons: “because I was just typing things onto a computer”, “because the computer chose the scenes for me”, and “because I was dragging blocks not typing”. The rest of the responses were uncertain or blank.

The common theme from affirmative responses here was clearly the interactivity of KW. Even with the relatively restrictive requirements to solve most lessons, students still felt they had



agency over how their code/story was authored. From the negative responses, students interestingly seemed to focus on the input method as being what separated KW from storytelling. They may have interpreted “storytelling” in the question prompt as something purely verbal, or maybe handwritten. Only 8 of the 15 students that answered no to this question also answered no to the previous one, meaning that while responses were more or less evenly divided for each question, the clear majority of students felt participation in KW resembled some aspect of the storytelling process.

Next we asked students to compare coding in KW to either playing or making a game. We did not specify a video game in the prompt but based on responses most students assumed this. In regards to playing a game, 22 students said it at least partially resembled playing a game to them, for reasons such as “because it was just as fun as one”, “it felt like kind of a mind game”, “because you were doing it on the computer”, “because in video games there is code!”, and “just like Minecraft”. Eleven students answered in the negative. Most in this group did not elaborate but one student mentioned they felt games needed to use a controller, and another student simply said “because it is coding”.

Lastly, in regards to KW feeling like making a game, 20 students answered yes with reasons like “because your coding someone to move”, “because making a game includes a LOT of coding”, “because I was programming something”. Fourteen students answered no for reasons such as “because I don’t know what making a game is like”, “because i was really bored”, “I still felt like I was making her life”, or “because not enough typing!”.

Unlike the storytelling questions there is a clearer majority here in favor of affirmative answers. This may be because storytelling can occur in a variety of mediums, but video games exist only on electronic devices. Students could have had different interpretations of what “storytelling” could mean but there appear to have been fewer ways the children interpreted the word “games”. Some students also mentioned games are represented in code, which makes comparisons to KW even easier. No student drew a clear distinction between programming interactive software vs.

programming a game, though some students suggested a game is identified by how fun it is to either play or create.

## 5.4 Student Code Experiences

As part of the quantitative data collection, both KW HAWK and KW Class captured each individual action students made as they edited their code, as well as the full program structure. These sets of programs are, to our knowledge, the first captured programming experiences of a group of recognized dyslexic children. In examining the progression each student took through the activities we have identified several behavior patterns we saw either repeated across lessons and individuals or were distinctive enough to be worthy of note.

The notion of identifying programming patterns has precedent in work examining “code smells”: patterns and behaviors in programming that are not explicitly incorrect but are still indicators that perhaps the program should be refactored (Fowler 1999). Others have demonstrated that these smells can occur in blocks code too (Hermans et al. 2016). While some of the patterns we discovered could be described as “smelly” the nature of code smells is that they are indicators of poor design and coding practices in general languages. Kokopelli’s World involves predefined activities with explicit goals, letting us label a program as “incorrect” with a degree of certainty that goes beyond how smells were originally intended to be used.

Preliminary work has been done in the Scratch language to use statistical analysis to identify when certain code patterns are occurring and use them to predict other behavior patterns (Dasgupta and Hill 2017). The two patterns featured in that study were related to variable initialization and object collision, neither of which are possible in Kokopelli’s World because of our design decision to avoid variables.

In the following section we describe patterns of coding behavior we observed ourselves. These include patterns we considered indicative of negative as well as positive behavior. These are

all listed below in general terms, and we have attempted to name them to better summarize them. After listing the patterns, we will follow the progression of a small number of example students by describing their behavior and progress and providing specific examples of the patterns as they occur. We offer these to provide potential insight into how dyslexic students learn programming and the pitfalls that can occur.

#### 5.4.1 Coding Patterns

**Lose-the-loop:** In cases where students are tasked with using a loop there are several ways, we saw students fail to apply the concept correctly. In one of those ways, students would construct what was functionally a sequential set of commands but wrap some or all of these commands in single-iteration loop blocks. We suspect, in their mind, this satisfied the requirement that their program use loops while still letting them think in terms of familiar sequential commands.

**Ignore-the-loop:** This is another variation of loop misinterpretation. In this case the student ignored the requirement to use loops entirely and just listed commands sequentially. In some instances, it is not clear whether the students failed to notice the loop requirement or chose to ignore it.

**Wrap-the-loop:** This is the last common variation on loops we witnessed, and a combination of the other two patterns in a sense. In this case the student constructs the desired behavior without any loop blocks, then puts the entire program within a single-iteration loop. This satisfies the loop inclusion requirement and typically performs correctly but does not demonstrate a proper understanding of loops.

**Ignore-the-if:** Taking the same idea from the similarly named *ignores-the-loop* pattern, in this case a student is asked to create branching behavior with a conditional block. They instead create commands of both branches of the loop and connect them sequentially with no control structures

applied. Within this pattern the student demonstrates that at least they know what needs to be done in the program, but not how to structure the contingency on the condition.

**Shuffle-the-if:** In this pattern we saw students who correctly identified the commands that should be in their program, and that the program should contain a conditional, but placed commands in incorrect branches of the conditional. For instance, putting a command that should come before a conditional in the conditional's true branch. We labeled a program with this pattern when at least two correct commands appeared in two incorrect locations around the conditional block, either before or after the block or in the true or false branches.

**Wrap-the-function:** This is similar to the *wrap-the-loop* pattern. In this case the student seems to fail to identify the usefulness of abstraction that functions can offer. The student constructs a full, sequential solution to the lesson then puts it in one function. There may or may not be a call to that function, but only at most one if there is.

**Special-event:** This is a general pattern we saw in a few instances. When the lesson asks students to perform some distinctive action in the latter half of the program - for example when Kokopelli should grab something in the middle of moving from one place to another - they will sometimes create the related command very early in the process, then set it aside until it is ready to be added to the main program body. This pattern suggests the student knows a distinct command is required later and creates it early to allow them to focus on other aspects of the program.

**Sequential-build:** Most students observed built their program one command at a time and attached them in chronological order of how they wanted it to be executed. This pattern was reserved for students that, in particular, ran their programs after each individual command was added. Students

following this pattern perhaps had trouble visualizing in their minds what each new additional command would do within the microworld and had to see it for themselves.

**Backwards-build:** Commands in Kokopelli's World were intended to be build left-to-right in the same order they are read. However, since blocks can exist disconnected in the KW programming space it is not a strict requirement to make commands this way. In this pattern students were observed creating commands in a different order; not creating a subject block first.

**Deja-fait:** This name comes from the French phrase for "made again". This is a small, simple pattern we saw often, in which a student would create a new block, immediately delete it, then immediately create another instance of the same block and add it to the program.

**Early-discovery:** Within each lesson in KW the block palette is restricted to just commands that are relevant to the lesson goals. Program structures are not restricted in the same way however. With this pattern we saw students demonstrate code patterns, either through accident or foreknowledge, that had not been explicitly taught in KW yet. For example, a student nesting two loops together before any lesson has required or mentioned this idea.

**Split-the-end:** In instances where students were asked to perform an action after a branching control structure had finished executing, they would sometimes place copies of this command at the end of each branch, instead of conserving code and putting one copy after the control structure scope. We called this *split-the-end*. If applied correctly this pattern would not produce incorrect code, just inefficient code. The KW static code checker made no distinctions on code style, so students would not receive any feedback, positive or negative, by using this pattern.

**Every-other-line:** Rarely we noticed a student making twice as many subject blocks as needed and using them to create “gaps” in the code. A student would use the blocks as spacers between the lines with actual commands. The equivalent to this in a textual language might be to double-space every line of code, though that is not quite the same since subject blocks are not the same as empty lines. This construction is allowed and does not affect the code behavior, but seemed to be aesthetically pleasing to the students, or they might have found it easier to read.

**Subject-stack:** This is a distinctive form of planning somewhat unique to KW. We occasionally saw students create and attach multiple subject blocks together before adding any other blocks or creating any complete commands. Other popular blocks-based programming environments do not use the same <subject> <verb> <object> format so we cannot easily draw an analogue to this behavior in other environments. We suspect they did this for convenience primarily. Rarely did students create the exact number of subject blocks that would be needed in the final program, so this is unlikely a demonstration of advanced planning on their part. Instead it is more likely they knew in advance a large number of subject blocks would be required and so made a number of them upfront to save time and attention later on.

#### *5.4.2 Pattern Rationale*

Before we give actual examples from students it is worth discussing how and why these patterns came to be. The above patterns all came from observing the programs produced by students and the processes they used. The control structure patterns related to loop, conditional, and function blocks resulted from our attempts to isolate and identify consistent, incorrect behavior seen in the students’ programs. While there were plenty of other programs with conditionals that were incorrect for a given lesson, we were not able to identify other patterns that represented a sufficiently large enough number of them. Since all three control structure blocks in KW define a program scope there were certainly parallels that can be drawn between how students used each block. The *ignore-the-loop*,

*ignore-the-if* and *wrap-the-loop* and *wrap-the-function* patterns being prime examples of this.

Likely because functions were given sole attention in their lessons and came much later in the series, we did not see instances of a hypothetical *ignores-the-function* pattern.

The *wrap-the-loop* and *lose-the-loop* patterns are closely related but appeared distinct enough to warrant a division between them. Both describe instances where students were trying to incorporate loops into their code but failed to do so correctly. In the case of *wrap-the-loop*, this is more easily identified as when students built a program how they would have before the introduction of loops, then trivially placed the whole thing into a one-iteration loop. If their program happened to be one command only, then this could also be an instance of *lose-the-loop*. To avoid ambiguity, we did not classify a program with a single command in a one-iteration loop as either pattern. Instead, *lose-the-loop* occurred when students were incorporating multiple loops into their program, but each loop was not contributing anything to the program behavior. This more often occurred when the lesson instructions stated that the solution required multiple loops as a hint, but students did not understand well enough to take advantage of it. We cannot say whether these two patterns represent distinct misconceptions of loops or not, but visually they were differentiated enough to warrant separate patterns.

The *special-event* and *early-discovery* patterns came out of a desire to recognize when students demonstrated an understanding of the environment or lessons that was beyond what was expected of them. In the case of *special-event*, these instances showed a level of planning that was identifiable in a way that was not apparent in many students' programs. Instances of *early-discovery* patterns were limited by the blocks palette within each lesson that restricted the blocks available to just those that were necessary to solve that lesson. In other words, it was not possible for a student to show understanding of how loops worked before they were introduced in the instruction because they were never available in a lesson before that point. Given the prior coding experience of many of our subjects it is likely that more of them would have demonstrated *early-discovery* patterns if the blocks palette had not been so restrictive.

The rest of the patterns, namely *deja-fait*, *split-the-end*, *subject-stack*, *every-other-line*, *sequential-build*, and *backwards-build* developed organically from noticing when students employed strategies that were distinctive and/or frequent enough to be identified as patterns.

### 5.4.3 Aggregate Pattern Counts

In this section we present and discuss a tally of the total number of occurrences of each of the design patterns, demonstrated by dyslexic students, defined earlier in this chapter. These results are in Table . These are not meant to be quantitative results but to give a broad sense of how prevalent each of these patterns are. We note that three dyslexic students in this set were involved with KW Class instead of KW HAWK. These students went through on average only a third as many lessons as the KW HAWK students did, and so are listed separately in their own column. They also did not have access to any lessons about functions, making any demonstration of the “wrapping the function” pattern impossible for them.

*Table 5.3: The number of observed occurrences of the previously described code patterns for all dyslexic students in their respective studies. Data for students that did not complete all lessons are included.*

Pattern	Occurrences in KW Class (n=3)	Occurrences in KW HAWK (n=17)	Total occurrences	Average per student
Lost-the-loop	2	10	12	0.63
Ignore-the-loop	2	7	9	0.47
Wrap-the-ooop	1	11	12	0.63
Ignore-the-if	2	11	13	0.68
Shuffle-the-if	15	7	22	1.16
Fill-the-if	6	14	20	1.05
Split-the-end	0	2	2	0.11
Wrap-the-function	N/A	12	12	0.63
Special-event	1	1	2	0.11
Sequential-build	0	14	14	0.74



Backwards-build	25	42	67	3.52
Deja-fait	73	276	349	18.37
Early-discovery	3	9	12	0.63
Subject-stack	5	21	26	1.37
Every-other-line	0	1	1	0.05

*Deja-fait* and *backwards-build* patterns occur at a level of granularity separate from the other patterns, as there is no limit to how many times they can occur in a lesson. This explains the much higher occurrence rate. It is difficult to conclude what students were thinking when performing either of these patterns.

Outside of them, *subject-stack* was the most popular pattern. *Shuffle-the-if* was also quite common, happening more than once on average per student. *Sequential-build* was the most distinctive pattern, in that each student using this pattern did so multiple times, but only five students in total demonstrated it. *Special-event* and *split-the-end* were not very frequent, being shown only once each by two students. *Every-other-line* was even less frequent, appearing only once in the whole data set.

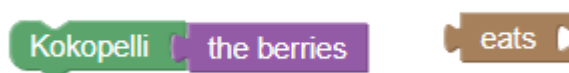
#### 5.4.4 Detailed Student Code Experiences

In this section we follow the progression of two dyslexic students and one typically developing student through their time with KW, highlighting any instances of the previously mentioned coding patterns or other unique behavior. These three students were chosen from the KW Class group rather than KW HAWK because the former group had higher event granularity. A third dyslexic student participated in KW Class but that experience is left out of this section because that student did not finish the last two lessons. Names used for these students are pseudonyms to protect their privacy.

## Dave

Dave is a third-grader, age nine, who had previously been diagnosed with dyslexia. He scored a 14 out of 24 on the second half of the reading test, which puts him at approximately the total average test score (13.68). Dave starts out the coding lessons well. Unprompted, he discovers by lesson three that commands can be stacked on top of each other, showing an *early-discovery* pattern that the KW system would not formally introduce until lesson seven.

In lesson six, which introduces the subject-verb-object command construction pattern, Dave attempts to connect a subject block directly to an object block, making a “Kokopelli the berries” command. The system does not allow these blocks to snap together, though the pieces do line up so with careful placement it is possible to make them *seem* like they have connected together, as shown in Figure 5.4. Dave might have been trying to see if the system recognized this construction anyway. When this program fails to run Dave quickly fixes this mistake however and moves on.



*Figure 5.4: Student Dave's attempt to create an incorrect command by carefully placing a Kokopelli subject block next to an object block.*

In lesson eight Dave is asked to make a lesson with two stacked movement commands. Commands only stack on the subject block. When creating his program Dave makes one command then creates a verb block for the second command, demonstrating a *backwards-build* pattern. Since the verb block has no way to connect to the first command he must put that block down, then make another subject block and connect it to the first command before adding the second verb. The point at which Dave had to set down the second verb is shown in Figure 5.5.

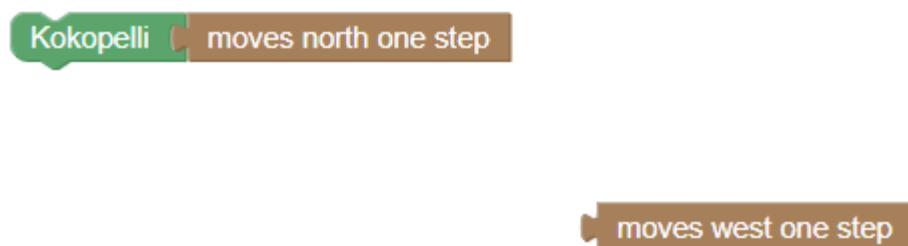


Figure 5.5: Dave's program when he tried to make a second command "verb-first" but had to put the second verb block down unconnected.

In lesson nine Dave performs a *backwards-build* pattern again. He tries to create commands verb-first or object-first before realizing they will not fit anywhere without a subject. He also tries to create a "Kokopelli the berries" command again by carefully placing them together. In this lesson he demonstrates a *special-event* pattern, as he knows at some point during movement Kokopelli must grab some berries. He creates the parts of that command early on but builds his program with more movement commands before the grab is finally added to the main program. He also leaves the verb without a subject for several steps. Eventually he does realize a grab command requires an object and fixes his mistake.

In the next lesson Dave immediately makes a correct grab command without issue, demonstrating he has learned how to do so. In this lesson, 10, he creates a few subject blocks before adding anything else to them, showing a *subject-stack* pattern in Figure 5.6. He even completes the command that starts with Raven before completing the one that starts with Kokopelli, suggesting he had a clearer understanding on how the program needed to end than what happened before that point.

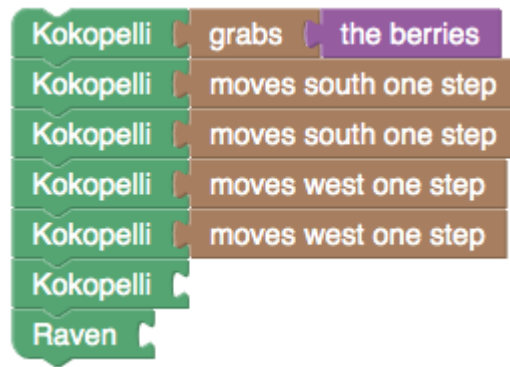


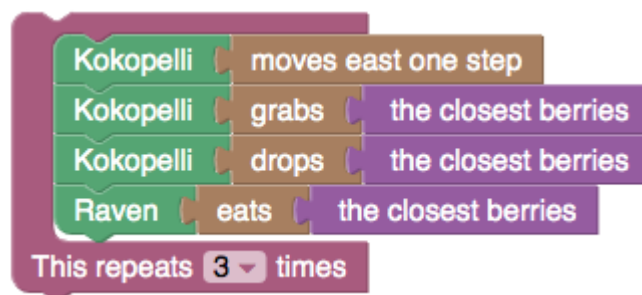
Figure 5.6: Student Dave's example of the subject-stack pattern, where he creates two incomplete commands with Kokopelli and Raven subject blocks at the bottom.

Lesson 11 is the introduction to control structures with loops. Dave appears to grasp the concept quickly, only showing hesitation when he creates, deletes, then creates a verb block again in a *dejai-fait* pattern. In lesson 12 he creates a command verb-first again in another *backwards-build* pattern but completes the whole lesson quickly.

In lesson 13 Dave must create a loop with multiple commands stacked inside. He quickly produces a correct loop with one command but struggles to figure out how to integrate the second. He first tries creating a subject-object command again. When that doesn't work, he switches which command sits in the loop, which does produce different behavior but is still not correct. Only after trying both of those does he add a second subject block after the first, allowing him to fit a complete second command in the loop.

Dave shows he understands the concept from the previous lesson by immediately making a loop with two stacked subject blocks in lesson 14. He completes the first loop command quickly but makes and deletes several blocks while picking what should go in the second before finally settling on the correct construction. This lesson introduces students to the concept of putting commands after a loop. On Dave's first approach he puts the commands that should come at the end of the loop inside it instead, as seen in Figure 5.7. When Dave runs the program, it is clear this is not the

solution. He tries another variation with one command removed, before finally putting the necessary commands after the loop.



*Figure 5.7: Dave’s attempt to solve lesson 14. The last two commands of the program need to be outside of and after the loop for it to be considered correct.*

In the first lesson where Dave is tasked with making a conditional, lesson 15, he correctly constructs the condition. He also puts the correct command into each branch of the conditional, though in the process he creates and deletes several commands before settling on the right one.

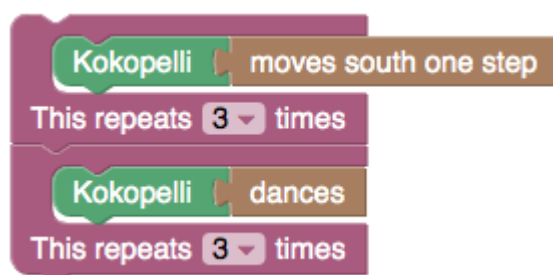
In the next lesson, 17, he must also make a conditional, but put a command before and after it. Dave creates the condition and branches correctly but struggles with what must come before and after it. He seems to interpret this as a mistake in his conditional and begins to unnecessarily edit it. He tries putting the command that should come after the conditional into both branches. When that doesn’t work he puts the command that should come before the conditional into both branches instead. Only after trying all of that does he attempt, correctly, to put a command before the conditional. From there he quickly grasps that commands can come after conditionals too and completes the lesson at that point.

Dave approaches the next lesson more carefully. Here he is asked to create several commands, some inside of conditionals and some outside. This is also the first lesson students are asked to make multiple conditionals. Dave creates and deletes many blocks while deciding what should be added to the program. After some trial and error he identifies and creates the first two necessary commands that occur before the first conditional. From there he creates the rest of the

program in the same order it will be executed. At one point he seems to struggle with the idea of a conditional having an empty branch but settles on leaving it empty. He very nearly finishes the program correctly but mistakenly makes Kokopelli eat the berries at the end rather than Raven. This causes him to backtrack a few extra steps and re-make the second conditional, but this time he uses the correct subject and finishes the lesson.

In lesson 19 Dave is asked for the first time to mix conditionals and loops. In this case, he must execute a conditional in a loop. Dave quickly fits the conditional in the loop and recognizes that a command should also be included in the loop but outside of the conditional. After that realization, he finishes the rest of the lesson without issue. Lesson 20 is a modification of lesson 19, where the same general program structure is required but students must add some more commands after the loop finishes. This time Dave reconstructs the program more confidently and adds the last two commands after the loop without any trouble.

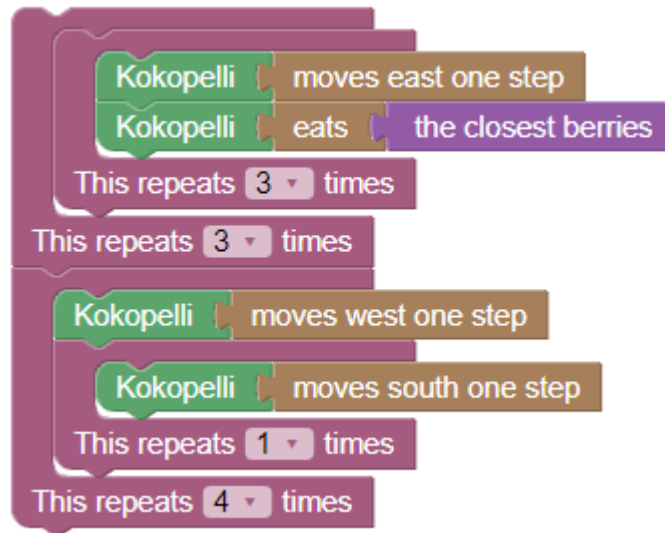
Lesson 21 is the introduction to nested loops. Dave creates one loop and command pair successfully, and then the other, placing them in sequence instead of nesting them. His program can be seen in Figure 5.8. When this fails to pass, Dave spends some time manipulating the blocks to eventually reach the nested structure that allows him to run the program again, pass, and move on.



*Figure 5.8: Dave's attempt to get Kokopelli to dance three times each time after he moves in Lesson 21. This program requires nested loops to be completed correctly.*

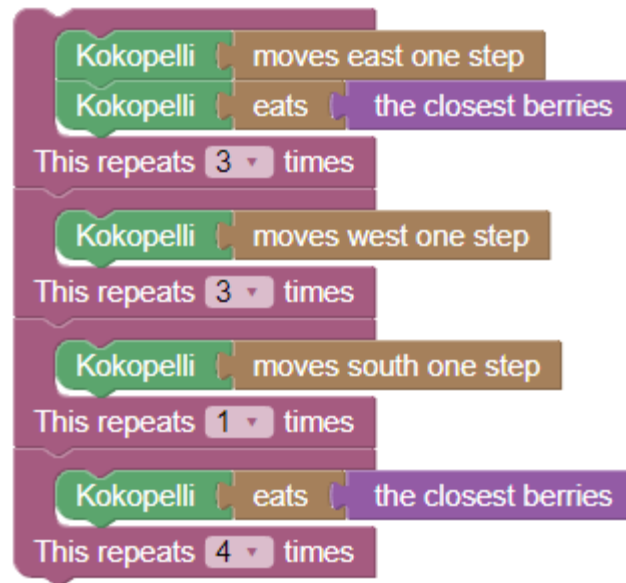
Lessons 22 and 23 were the hardest of the set, and so it was no surprise that Dave spends a lot of time in them. Both lessons tasked moving Kokopelli to every cell of a grid by moving left to

right, top to bottom in a scanning pattern. This required two loops nested inside of a third and frequently stumped students. Dave starts well by nesting two loops together. In the inner loop he has Kokopelli move east and eat berries, both of which are correct. He then adds another loop at the end with Kokopelli moving west. Then another loop inside that with Kokopelli moving south.



*Figure 5.9: A program from Dave's attempt at solving lesson 22. He uses a superfluous loop around the move south command, and incorrectly separates the movement into two outer loops.*

When this program fails, Dave starts to demonstrate a *lose-the-loop* pattern. He begins to add more loops, most of which don't change anything, and then changing directions of commands to see what results. When this also fails he tries a different approach and removes most of the extra loops but leaves out the outer loop as well.



*Figure 5.10: Another snapshot of Dave solving lesson 22. In this instance he has nearly the correct behavior but is missing the final outer loop. After this point he will try several more programs before discovering a solution.*

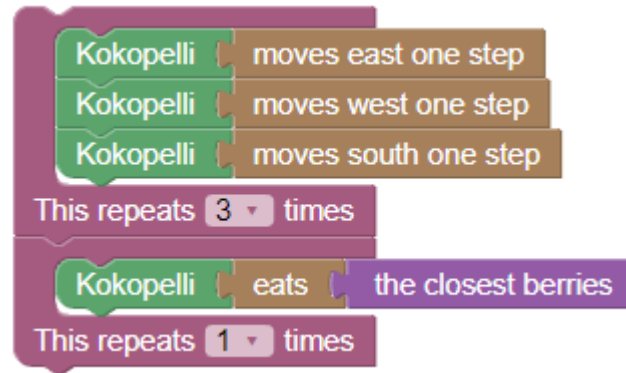
He appears happy with this approach, and so starts to repeat it, showing an instance of *ignore-the-loop* as he begins to copy commands. In short order Dave's program, shown in Figure 5.11, has gotten long enough it doesn't fit on the screen. Seemingly at this point Dave realizes something is off and he deletes the entire program and starts over.





*Figure 5.11: Another program from Dave’s attempt at solving lesson 22. His program is correct for one iteration, but instead of using an outer loop he has started to create a second copy of the desired behavior, demonstrating a lose-the-loop pattern.*

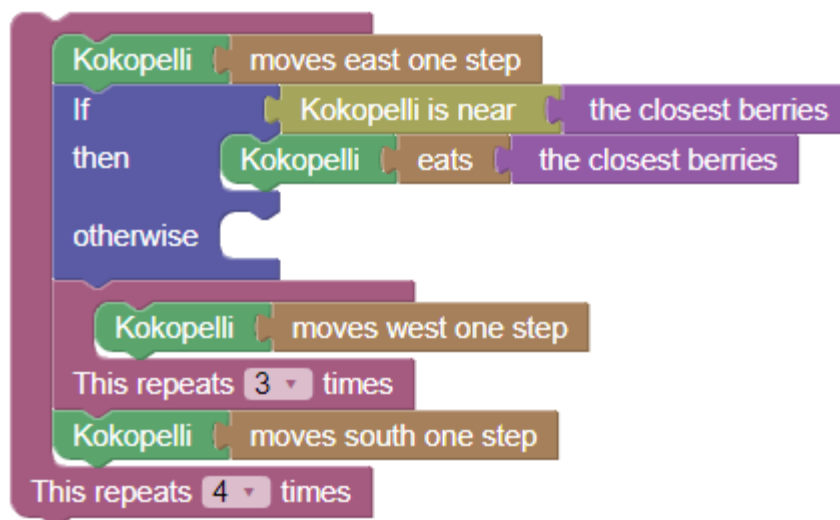
He begins by constructing the contents of the inner loop, mostly correctly with one superfluous loop. At this time Dave’s class session ends and he returns several days later to start again. In Figure 5.12 we can see he this time tries to fit all relevant movement commands into the same loop, then another loop to handle when Kokopelli eats berries. This also fails to pass but his next attempt is much closer. He separates the movement commands into their own loops and adds eat commands to each loop as well.



*Figure 5.12: Another common incorrect solution for lesson 22 demonstrated by Dave. His program contains all the necessary blocks for one complete iteration, but he has ordered all his movement commands incorrectly.*

This is unnecessary but not incorrect. Importantly, then he remembers to add the outer loop back in. With this last addition his program is correct and he is allowed to move on. All together Dave spends over 36 minutes solving this lesson, spread across two separate sessions.

Lesson 23 is very similar to 22 with the single addition of a conditional block. It is mainly there to confirm if students actually understood the goal of lesson 22 and would be able to recreate it with one alteration. Dave fares well here. He starts by creating the same initial loop and adds the conditional in the correct location and with the right construction. He creates a nearly correct program, missing only one inner loop, in Figure 5.13.



*Figure 5.13: Dave's incomplete solution for lesson 23. This program is nearly correct but is missing an inner loop around the east movement and conditional.*

He then goes to add another inner loop and struggles for a short while to place it exactly where he wants it. He does not run the program through this part so we assume he had the right idea but was struggling with the blocks interface. When next he runs the program, it is complete and he is able to move on, having spent 11.5 minutes on this lesson.

Lesson 24 take a different format than past lessons. Students have a clear goal, move Kokopelli through a maze, and are allowed to create any program they want to do this. They are not told what blocks to use but are encouraged to think about how few they could use to solve the lesson. Dave attempts to create the full program before running it. He uses loops to eliminate repeated commands. He realizes partway through that there is an additional requirement to the lesson, he also has to ensure Kokopelli grabs some berries on his way out of the maze. This requires Dave to splice one of his loops and add a grab command in between. Once having done this, Dave is able to complete the lesson and move on, finishing just shy of eight minutes.

Lesson 25 is entirely open-ended. Students are allowed to use whichever blocks they want to do whatever they want, and to move on they must simply run the program once. Most students

chose to move on immediately. Dave makes a single movement command before continuing on to the questionnaire.

Altogether Dave does not demonstrate great difficulty with the programming material. He shows positive patterns early on with “early discovery” and “special event”, indicating either prior experience or a quick understanding. He is able to recall skills learned in prior lessons, and does not repeat the same mistakes often, showing at most a few instances of each negative pattern. He has the most trouble with lessons 22 and 23, which is not uncommon, and his shorter time with 23 shows he was able to carry over information from 22. When left unprompted in lesson 24, he still chose to include loops in his program.

## **John**

John is a 4th grader, age 11, diagnosed with dyslexia. He scored an 11 on the second half of the reading test, which also puts him in the lowest quartile along with the third dyslexic student in KW Class. Though only lower than Dave by three points, John is fairly far behind Dave in the overall distribution. We cannot say if it is directly related, but John also struggles with the lessons more than Dave, taking 17 more minutes to complete all of the lessons.

Early on John demonstrates a preference for the *backwards-build* pattern, such as making a “move north one step” verb block before making and connecting a “Kokopelli” subject block. This continues for several lessons. In lesson six, where commands that use more than two parts are introduced, John continues to make commands in “backwards” order by creating “the berries”, then “eats”, and then “Kokopelli”.

Lesson seven is the first instance where users must connect two commands together. This is when John’s verb-first construction strategy causes a problem for him. The activity tasks users with creating and connecting two movement commands together. John constructs the first command verb-first like previous lessons, but when he creates a second verb block he realizes there is no place to connect it. Commands only connect via the subject blocks, which John has not made for his

second command yet. This also happened with Dave, but instead of putting the block down John responds by deleting his second verb block and creating a command subject-first for the first time.

In the next lesson John's construction strategy is more irregular. In this lesson he must also make two connected movement commands. For the first command he makes a verb first again, but changes his mind and deletes it then makes a subject instead. For the second command he makes a verb first but keeps it this time, dropping it alone on the workspace. He then makes another subject block, connects it to the first command, and then picks the verb back up and connects it to the rest of the program. This mixture of command build orders is uncommon. We more often saw students permanently stop using *backwards-build* patterns than occasionally return to them.

In lesson nine John must create his first complete grab command, which is another three-part command. This time he makes a grab verb block - a middle block - first. He then makes the subject block and connects it to the grab, a second grab block that he quickly deletes, and finally a berries block to act as the object and complete the command. The rest of the program is movement actions which John all makes verb-first. From this point on John's command creation order is irregular, with some commands being made subject-first and others verb-first.

In lesson 10 John constructs a drop command without an object and keeps it in his program for several actions and executions before adding an object, seen in Figure 5.14.

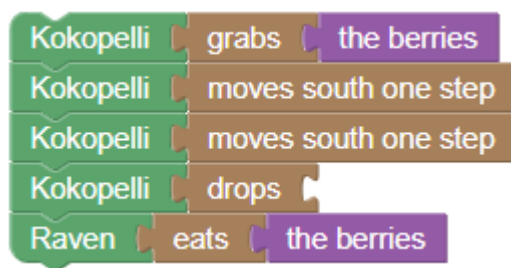


Figure 5.14: John's program missing a drop object block.

In lesson 11 John constructs his first loop correctly with no hassle, and also demonstrates an *early-discovery* when he connects a subject block to the outside of the loop before being prompted to do so.

Lesson 13 is the first instance where a student must stack multiple commands in one loop. John first tries to stack these commands before the loop block, as seen in Figure 5.15. When this doesn't work he moves them into the loop correctly and proceeds.

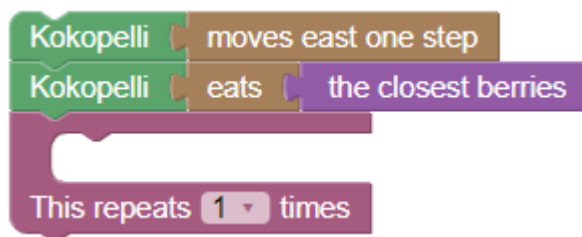


Figure 5.15: John's program with commands in the incorrect location (outside of the loop).

In lesson 14 John demonstrates a new strategy, where he makes all of the verbs that will become part of the program first before any other blocks. We consider these forms of the *special-event* pattern, the code for which is shown in Figure 5.16. He then constructs the program in a standard fashion and does not actually use the original verbs except for one command. The rest of the verbs are ignored and deleted when the main program appears finished. This lesson is also where John has his first difficulty with loops. The program solution requires two commands in a loop and two commands immediately following it. John tries one command in the loop, and when that fails tries all of them inside it which is also incorrect. He tries a few more combinations of commands inside and outside the loop before finding the correct one.

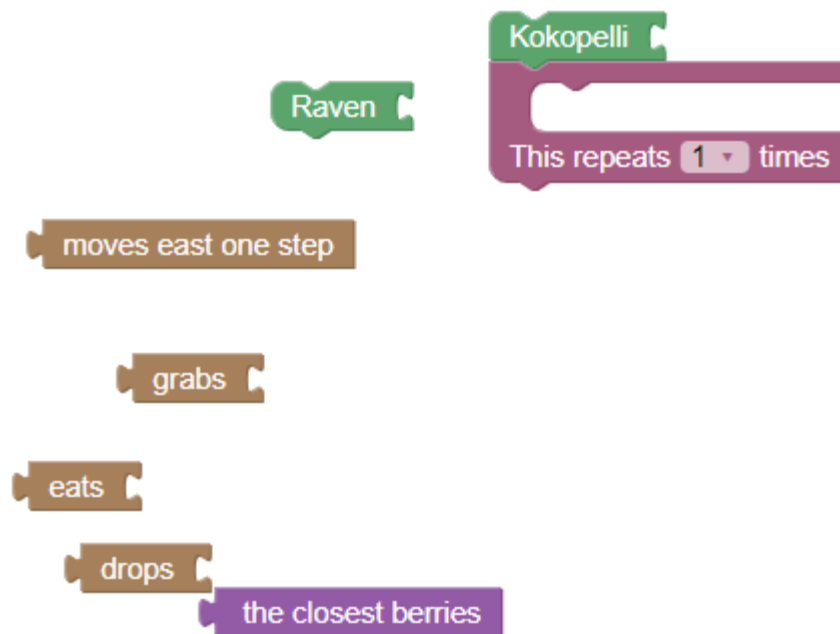


Figure 5.16: John's program where he makes all the verbs he intends to use in advance.

In the first lesson where John must make his own conditional, lesson 16, he demonstrates previous *backwards-build* habits by making his conditional with the object first and does the same for the commands that go inside it. Importantly, his actual program is still correct. Lesson 17 is similar but also requires an additional command to appear after the conditional. The student must make Kokopelli move north or south depending on the conditional, and then try to eat berries afterwards regardless. A common mistake for this lesson is to place the eat command inside of the conditional somewhere. John makes this mistake, and when that fails he expresses seeming frustration with the says block, shown in Figure 5.17. His program here is also an example of the *shuffle-the-if* pattern.

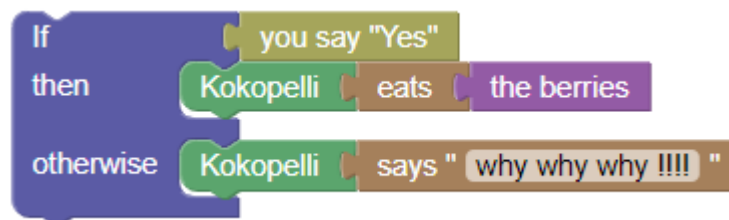


Figure 5.17: John's incorrect solution to lesson 17 where he expresses frustration.

John realizes movement is required in this lesson so he adds a moves north command before the conditional, then eventually into the correct branch. This is not sufficient to pass the lesson requirement, so he tries a few other permutations, demonstrating the *deja-fait* pattern in the process. John runs out of time at this point and returns to the lesson another day. Upon returning, John pieces together all but one of the correct commands but puts them in the incorrect order, with one movement coming before the conditional and the eat command in the true branch, shown in Figure 5.18. After building this program he rearranges the commands, adding the last, necessary dialogue block, and finally passes the lesson.



*Figure 5.18: Another incorrect solution for lesson 17. This one contains nearly all the correct commands but in the wrong order.*

Lesson 18 starts out well for John. This one requires him to use two sequential commands followed by two sequential conditionals. He makes the first two commands correctly, but then does an *ignore-the-if* and makes only the contents of the two conditional branches. It is not clear if he is interpreting the instructions as a description of the program execution rather than the program structure, or he is aware and this is a “trial run” so to speak. This program is in Figure 5.19.





*Figure 5.19: John's first attempt at lesson 18. It has the correct behavior but the last two commands should each be in its own conditional block.*

In either case, after running that program he deletes it and starts over, likely because it failed to pass, and makes a conditional block first this time. He constructs the second required conditional correctly but is still missing the first. When this program fails he seems to doubt his program and deletes all of it a second time. His next attempt includes the first two commands correctly, as well as the first conditional, but not the second. When this program also fails he adds the last conditional and passes the lesson.

In lesson 19 John must put a command and a conditional both inside of the same loop. He makes the conditional and loop correctly but places the command before the loop. It may be that John did not know at the time that multiple commands can stack with a conditional inside of a loop, though John works this out fairly quickly and moves on.

Lesson 20's format is similar to that of 19 but asks students to place two additional commands after the loop. John completes this lesson more efficiently than the last with fewer incorrect tries. He demonstrates a *fill-the-if* pattern we see in other students where one branch of the conditional is not specified in the instructions, but John fills it with a command anyway. This is considered acceptable but can result in unexpected program execution behavior.

John grasps the use of nested loops with no trouble and completes lesson 21 with no wasted actions.

In lesson 22 John starts strong with a nearly correct program for one iteration of the solution, however when this fails he scraps it all and starts over again. John tries various

permutations of loops and movement commands but it becomes apparent he is not clear on the intended behavior of Kokopelli, as most attempts are missing key movement commands or are clearly mis-ordered. After nearly 14 minutes since starting lesson 22 John creates a program that is very nearly correct but has an incorrect number of outer loops. John fails to notice this and spends a few more minutes re-ordering his code trying to find a solution. After 18 minutes he comes back to his old structure with the correct number of iterations and is able to progress.

Lesson 23 is only a small alteration to lesson 22 but takes John nearly twice as long to solve it. John appears to be unaware of how similar the two lessons are, as his early attempts do not resemble the solution from 22. It is difficult to follow his thought process for this lesson. He makes several attempts that fail, deleting some or all of the code after each try. His programs are incorrect and it is not clear if he is actually attempting to solve the problem or just experimenting with code. After 15 minutes he scraps his program once again and starts on what will become his solution. For the first time he incorporates a conditional block, something required in the solution. After a few iterations his code is much closer to the solution. In the final step he moves the conditional into an inner loop and solves the lesson.

In lesson 24 John demonstrates a fair understanding of nested loops, electing to use two of them early on in his solution. The rest of the program uses only single loops but does not contain any wasted or extraneous commands. John does not elect to make a program in lesson 25.

Overall John appears to have had more trouble with the lessons than Dave. He starts out well and his backwards build strategy does not initially get in his way. When control structure blocks are introduced though we see him struggle to connect commands together with the *backwards-build* strategy. John in fact uses the *backwards-build* pattern more than any other student we observed. Since every student we examined eventually stopped using the *backwards-build* pattern in later lessons, including John, his insistence may have made the transition harder for him.

While John also uses a *special-event* pattern like Dave, John's is less precise and the blocks he makes in advance don't actually end up in his final solution. John more often deletes his entire

program and restarts and demonstrates more control-block-related patterns than Dave. There are also fewer points in John's data where it is clear he learned something from a previous lesson. His experiences in lessons 22 and 23 are examples of this. Ideally students have an experience similar to Dave where they recognize that lesson 23 is asking for a variation of the solution from lesson 22 and start work from that point. John does not realize this and so spends significantly more time in lesson 23 than lesson 22. Even while John solves lesson 22 faster than Dave, because of difficulty transferring knowledge from lesson 22 he ends up spending more time overall on the two lessons.

One positive indication of John's learning is his unprompted use of nested loops in lesson 24. Dave did not use nested loops for this lesson so while John struggled more he was still able to effectively apply a relatively advanced strategy near the end of the lessons set.

### **Sarah**

Sarah is an eight year-old third grade student who did not report any learning disabilities in her entrance survey, unlike Dave and John. Sarah scored a 15 on the reading test, which puts her slightly above the average score (13.68) and slightly below the mode score (16). While each student is unique, Sarah is our attempt to show a representative sample of what a typically-developing student's experience with Kokopelli's World can be like.

Sarah begins her time in KW by almost immediately demonstrating an *early-discovery* pattern. On the first lesson where she can make any blocks at all, lesson two, she connects four blocks together vertically. She does this again on the next lesson as well. In lesson five Sarah tries to connect two verb blocks together - shown in Figure 5.20 - in an apparent attempt to make Kokopelli dance twice in one line. Since at this point Sarah has already demonstrated an ability to stack commands, this appears to be an attempted shortcut rather than a misconception about how commands are formed at all. In the same lesson Sarah continues to create and stack more commands than was asked for in the instructions, showing a *subject-stack* pattern in the process.

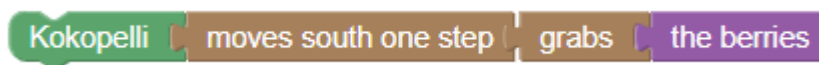
This is a theme that will continue through many of her lessons and from the capability shown in these early examples it appears to be a sign of boredom on her part.



*Figure 5.20: Sarah's program attempting to connect two verbs together in lesson five.*

In lesson six she performs another *subject-stack* to create four total commands, three more than the single command asked for. In lesson seven when command stacking is formally introduced, Sarah only creates the two commands requested. She performs the minimum for lesson eight as well.

In lesson nine Sarah tries to connect two verbs together again, this time in a more complex example that also involves a subject block (shown in Figure 5.21). Sarah then uses the verbs to make complete commands in a few instances of the backwards build pattern.



*Figure 5.21: Sarah's more complex verb stacking program from lesson nine.*

In lesson 11 Sarah quickly and correctly assembles her first for loop. Without running her program she proceeds to create a *subject-stack* and then generate a large number of consecutive "hit" commands, shown in Figure 5.22. Eventually her program holds 18 hit commands. These are all deleted after she runs the program once. It appears she just wanted to see what such a program would do, rather than she misunderstood the prompt. Sarah then finishes the program correctly and moves on.

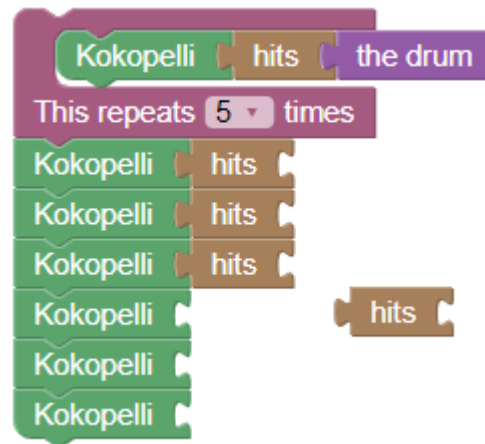


Figure 5.22: Sarah's program with a correctly-structured loop and superfluous commands afterwards.

Sarah's understanding of loops is corroborated in lesson 12 when she completes it quickly with nearly the minimum number of steps required. In lesson 13 Sarah generates a correct but peculiar program with repeated commands in a loop, shown in Figure 5.23. Due to the way the code checker is structured in Kokopelli's World, this program passes the test, since it contains all the required elements, but the second movement and eat commands are not needed. We did not classify this program as an instance of any loop-related code patterns either, as it is still technically correct.



Figure 5.23: Sarah's lesson 13 program with repeated commands that still passes.

This program may have indicated the beginning of a new strategy because in lesson 14 she demonstrates a *ignore-the-loop* pattern and tries to solve the lesson without loops at all, which are required. When this program is rejected she begins to incorporate loops again, though still repeats

commands needlessly inside of the loop. Such a program can be seen in Figure 5.24. Since Sarah initially appeared to be confident with her loops, this change in strategy is somewhat confusing. It is not clear if Sarah’s understanding of loops was only superficial, or if this is another case of her trying something different out of presumed boredom. At this point Sarah’s session ends and she returns to the lesson on another day. When she returns, Sarah restarts from scratch, makes another program with *ignore-the-loop*, then incorporates loops again. She tries different combinations of commands all nearly correct but missing one or two required aspects. In one instance it appears Sarah thinks berries can only be eaten if they are being held by a character, which is not the case and should be clarified in future versions of the instructions. Eventually Sarah finds the right combination of commands and proceeds onward.

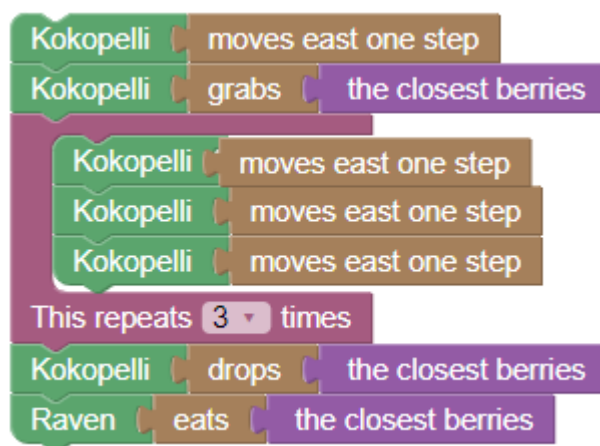


Figure 5.24: Sarah’s lesson 14 program which uses a loop, but does so incorrectly.

In her first lesson with conditionals, lesson 15, Sarah tries to incorporate a condition block into a command, making a “Kokopelli Kokopelli is near the berries.” This is not an allowed construct and the blocks do not properly connect. The only required actions in this lesson are to run the provided program, delete one command, then run it again, so Sarah’s decision to take the program apart and try different block combinations is noteworthy.

In lesson 16 Sarah has the opportunity to try making conditionals herself. She is quickly distracted by the introduction of the “says” block instead. Once she runs a program with it and realizes she can make Kokopelli say anything she wants she ignores the lesson requirements and tries a series of different text inputs, one example is shown in Figure 5.25. At this point the session ends, but Sarah continues to play with the block when she returns. When she actually tries to solve the lesson she quickly gets it done. This lesson is also a rare instance where a student solves the lesson, is given the choice to move forward, but continues to tinker for a while after. In this case she tries out a few more inputs to the says command.

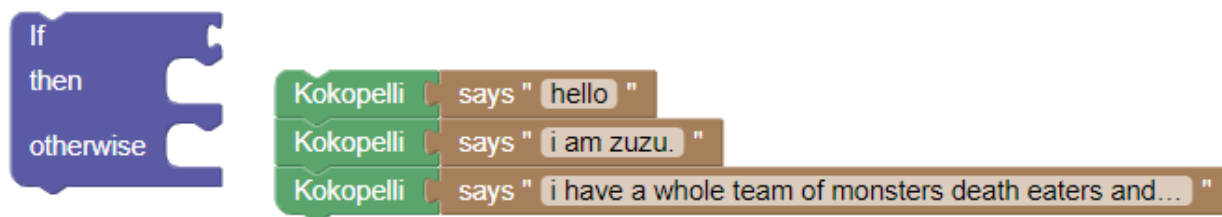
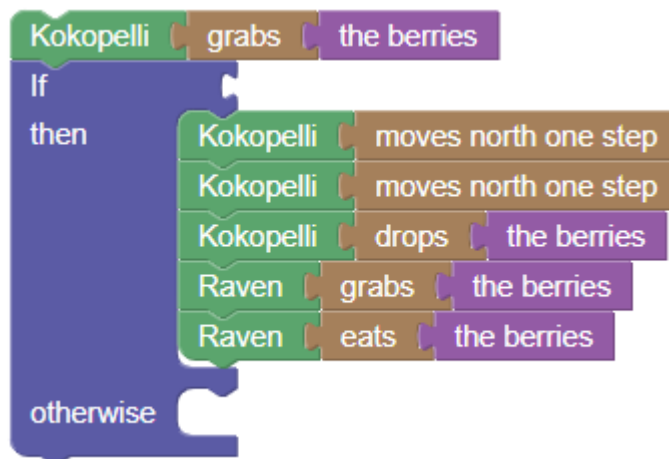


Figure 5.25: Sarah’s lesson 16 program where she ignores the required conditional and plays with the says command, complete with Harry Potter reference.

Sarah shows more focus in the next lesson, but still incorporates her own says commands for fun. She also appears to have more trouble actually solving the lesson showing a *shuffle-the-if* pattern in the process. This continues into lesson 18 where she does *shuffle-the-if* again, and constructs a peculiar program where she creates all of the described behavior and then puts the whole thing into a branch of an if block. Making some sort of combination of an *ignore-the-if* and *wrap-the-loop* but with conditionals. This is the only instance of such a program we observed so it was not included as a formal pattern, but it is shown in Figure 5.26.

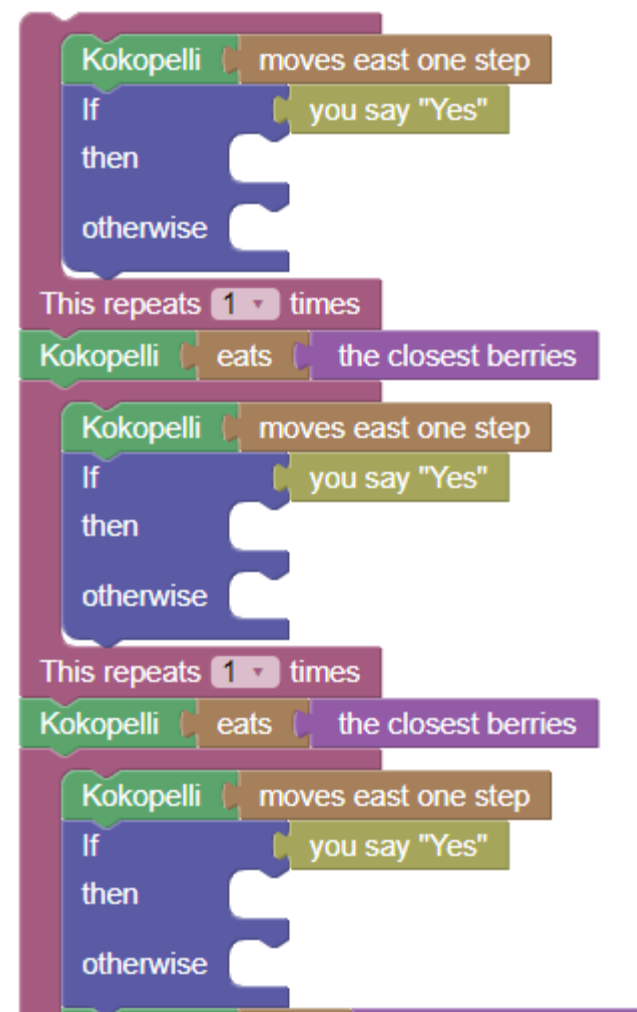


*Figure 5.26: Sarah’s peculiar pattern from lesson 18 that combines aspects of the “ignore the if” and “wraps the loop” patterns but remains unique.*

This program fails to pass, and just before the session ends Sarah tries another program with *ignore-the-if* again. When she returns Sarah tries this pattern again. She then shifts towards the solution by incorporating multiple if blocks but still goes through many incorrect permutations. In the process she demonstrates an early discovery pattern by nesting the conditionals inside of each other. After further tinkering and a full lesson restart she is able to construct a correct program and move on.

For lesson 19 Sarah must learn to use conditionals inside of loops. She tries many combinations of these and basic commands, and aside from the occasional lively “says” command appears to be on-task the whole time. During her experimentation she demonstrates several patterns starting with *fill-the-if*, then *shuffle-the-if*, then *lose-the-loop* twice once without conditions and once with, the last of which is shown in Figure 5.27.





*Figure 5.27: Sarah's program from lesson 19 that shows a variant of the "lost the loop" pattern which incorporates if blocks as well.*

As demonstrated by Dave and John, lessons 22 and 23 have shown to be some of the most difficult in KW, so it is surprising that Sarah finishes both of them quite quickly and with little trouble. Sarah finishes 22 in just under four and a half minutes with few erroneous programs along the way to a solution. She takes less than a minute longer to solve 23 as well.

For lesson 24 Sarah writes nearly all of her program before testing it, and only has to make a few small edits to reach a solution. Notably she does not elect to use any control blocks in her

program at all, including loops. For the optional lesson 25 she makes a single complete dance command before moving on.

While it is not clear if it is due to her lack of dyslexia, Sarah distinguishes herself from both John and Dave in numerous ways. Her very early and seemingly confident use of early discovery suggests she probably had more prior experience with blocks than John or Dave. This might have been to her detriment though as she went off task much more frequently than the other two students. This case shows one of the potential issues with using time taken in lesson as a measure of success. It is not clear if time spent off-task should be treated with the same weight as time spent struggling while remaining on-task. In total Sarah spends nearly two hours to complete all of the lessons, which is 15 minutes more than John and more than a half hour longer than Dave.

Perhaps explaining the longer time, Sarah's initial advantage appears to have diminished with the later lessons. Her time spent on task increases and she starts to perform negative coding patterns more similar to John and Dave. This is in sharp contrast to her performance in lessons 22 and 23 though. She solves these so efficiently that we suspect she may have received help from a classmate or instructor. Curiously, right after these lessons Sarah does not elect to use any control blocks at all to solve lesson 24. Perhaps she did not feel the need to use control blocks when it wasn't required. Her decision not to run the program until it was nearly finished shows she at least did not have trouble maintaining a mental model of the program's execution, though maybe she felt adding control blocks into that model would have been too taxing.

#### *5.4.5 Individual Student Tallies*

For comparison purposes we now share the individual counts of coding patterns for the three students introduced in the previous section. The individual tallies of coding patterns demonstrated by Dave, John and Sarah are shown in Table 2.

Dave did not demonstrate an abnormally high number of occurrences for any pattern, but still showed a preference for some over others. He had one instance each of *lose-the-loop* and

*ignore-the-loop* patterns, but none of *wrap-the-loop*. He also had two instances of *shuffle-the-if* and no instances of *ignore-the-if*. This suggests Dave was not reticent to use if blocks even in instances where he used them incorrectly. He also correctly identified situations where an if block was required.

Table 2.4: Coding pattern tallies for the three featured students, along with the averages over all dyslexic students.

Pattern	Dave's Patterns Tally	John's Patterns Tally	Sarah's Patterns Tally	Average per student
Lose-the-loop	1	1	2	0.63
Ignore-the-loop	1	1	2	0.47
Wrap-the-loop	0	1	1	0.63
Ignore-the-if	0	2	2	0.68
Shuffle-the-if	2	4	3	1.16
Fill-the-if	1	4	1	1.05
Split-the-end	0	0	0	0.11
Wraps-the-function	N/A	N/A	N/A	0.63
Special-event	0	1	0	0.11
Sequential-build	0	0	0	0.74
Backwards-build	3	17	4	3.53
Deja-fait	6	7	11	18.37
Early-discovery	1	0	2	0.63
Subject-stack	2	0	4	1.37
Every-other-line	0	0	0	0.05

In contrast to Dave, John shows a greater predilection for demonstrating patterns related to control blocks. He has the same number or more of every pattern except *split-the-end*, including a notably large four instances of *shuffle-the-if*. With ten total instances of conditional-related patterns it would seem John had more trouble than Dave in navigating that set of lessons. John also demonstrated the highest number of *backwards-build* instances among all students measured at seventeen.

Sarah's individual tallies show that her initial apparent familiarity with blocks did not prevent her from demonstrating several negative patterns when it came to the complex control-structure-block-related lessons. Like John she demonstrated at least one instance of each control-block-related pattern except *split-the-if*, and more in some individual categories than John. Sarah likely best distinguishes herself with the largest number of *subject-stack* and *early-discovery* instances we measured (tied with three other students and two other students respectively) at four. In her case the *subject-stacks* were attached to apparent instances of boredom or off-task programming, which was not the case for the *subject-stack* instances observed in other students. While she did show an early few instances of *backwards-build*, Sarah stopped after only four times in two lessons, which sets her apart from John who continued for many more. Overall Sarah demonstrates that typically developing students do not necessarily show fewer instances of negative patterns. Her particular tallies do clearly differentiate her from Dave and John however.

#### 5.4.6 Interview Responses

We now share the most relevant responses from the pre and post interview questions of the two dyslexic students that participated in one-on-one sessions. Their pseudonyms will be Katy and Will. The full list of interview questions is shared in Appendix D and are divided into pre and post-treatment sections. The pre-interview questions are largely concerned with gathering a more detailed sense of the student's prior programming experience and their thoughts on it so far. The post-interview questions ask students their thoughts on the unique KW features and probe their understanding of the programming concepts taught in KW.

#### **Katy**

Katy is a dyslexic 6th grade student, age 11, who scored a 12 on the reading test, putting her between Dave and John's level on that particular task.

Katy had a fair amount of programming experience before the study, having programmed a Finches robot using the Scratch and Bitsbox languages to help physically disabled users pick up objects. This happened in-class over a semester-long period. Katy enjoyed her experience with the robot and the sense of accomplishment that came from planning the robot to do something and then executing that: “I liked learning how to do it. It was fun learning how to make it do something and then, like, having it do it. Like that was an accomplishment, like ‘Wow! It moved!’ and I made that happen.” Katy expressed some frustration with using the Finches, but these were logistical issues and did not appear related to her dyslexia. She was excited at the idea of working with the Finches more in the future, and a little more reserved but still interested in programming in other contexts.

As an informal pre and post-test we asked students, in general terms, if they were familiar with the control concepts of loops and conditionals. When asked if she was familiar with the concept of a repeating loop, Katy did not recognize the concept. She was able to draw some connection to the concept of a conditional, but her experience with them in Bitsbox appeared to have been greatly abstracted.

When asked about her idea of what professional programmers do in their jobs, Katy thought of programming and controlling robots. She thought they would use something “higher level” than the Finches robots that was more high-tech or they would make their own robots instead. If given the opportunity to make anything she wanted with programming, Katy would have liked to work with robots more.

After participating in the reading test and KW lessons we asked Katy a verbal version of the same questionnaire that typical KW Class students answered. Katy liked the lessons, and most enjoyed using the repeat block. She thought the audio in the system was useful, but “wished it were a little faster.” She thought programming the Finches from before was easier than using KW, but enjoyed the challenge of KW. She was open to doing more lessons in a KW-like environment. Using KW did not change her general view on programming. Katy thought using KW was more

like being read a story than telling one, but saw some similarity to both ideas. She did not think that using KW felt like playing a game, but guessed that it felt more like making a game.

After the regular questionnaire we also asked Katy additional questions going into more detail about her experience with the system. Katy did not have any unanswered questions about the blocks in KW, and thought she understood how they all worked. In her own words, she described the repeat block as “it repeats another block so that you don’t have to repeat the same block fifty times”. When asked again if she recognized the concept she now thought she had seen something similar in her experience with Finches.

We asked her similar questions about the conditional block. She described the function of the conditional block as “step-by-step stuff like, do this and then do this and if that doesn’t work then do this.” She did not recognize this concept from her experience with Finches. She thought the lessons taught her how to use the conditional block sufficiently, but said she learned mainly by experimenting with it. One difficulty she mentioned with learning to use the conditional block was that not all blocks she expected could connect to it.

Katy most enjoyed lesson 24, which felt “more like telling a story” and “was kinda like a video game too”. Lesson 24 is an open-ended lesson where the student must guide Kokopelli through a maze in any way desired. Katy least enjoyed lesson 14, which she had great trouble with. This lesson involved using two conditionals in series to tell a short story of Kokopelli feeding Raven. She thought it was difficult because she was thinking “out of the box and I was supposed to think inside”. When asked to further explain she said “I think like, ‘how does this...what am I missing?’ I don’t really think ‘oh what blocks do I already have?’” Anecdotally many students have shown to have trouble with this lesson, and it is possible the instructions are unclear and should be edited in a future version.

After they were added to the blocks in the experimental condition, Katy did not use the speaker icons for additional audio support. She did not feel that she needed them. She used the “Say My Code” output option once during the lessons seemingly by accident but did not remember using

it or its effect when asked. She did notice when the code started to be read by the computer and thought it “made [understanding the program] a little bit easier”. “Reading [the code] twice” is what helped her better understand the code. She did not think a similar feature added to Finches would have helped because it was already clear what the robot was doing from its actions. When asked about the difference between the Finches robot’s movement and Kokopelli’s movement in the microworld, Katy said the Finches robot uses arrows to indicate movement which she thought was clearer than the cardinal directions used in KW.

Katy saw more similarity in KW to reading and writing than to math because “you have to read a lot to understand the directions and you have to write the code.” Katy thought KW was more likely to resemble what profession programmers do than Finches because it was more complex.

## **Will**

Will is an 11 year-old 5th grade student who had previously been diagnosed with dyslexia and scored a 6 on the reading test, which tied him with the lowest scores we measured among the classroom students.

Will also came into the study with prior programming experience. In his case he had made games in Scratch during the prior school year, including games like Space Invaders, for roughly four hours. He enjoyed his time with it, particularly because of the community that promoted shared code, but he thought moving code from other projects was difficult. “There’s, like, different, like, little boxes and you would have to write stuff or, like, read it and then move it - because there’s a lot of them - move it into the space that you needed it to be.” He was interested in working in Scratch more in the future.

Will recognized the concept of a repeating loop but was not able to clearly articulate its representation in Scratch. He did not recognize the concept of a conditional. Will thought of professional programmers as people who design and help code video games. He also described someone who would write tests for software. He did not think professional programmers used

Scratch or anything like it. Given the opportunity to make anything with code, Will wanted to make a first or third-person shooter game “like Fortnite.”

After going through the lessons Will thought he had a good grasp of all of the blocks used in Kokopelli’s World. Will remembered using repeat blocks from the lessons, and when asked to describe their function he chose to use an example. “The one [lesson]where it was like, move then dance, move then dance...Instead of doing all of the stuff, you could have repeated the move then dance however many times you needed to move.” Will recognized the repeat concept from his experience with Scratch but had not used it extensively. Will also recognized the conditional block from his past experience with Scratch, and from his work in KW. He described it this way: “You have the if, and then if he was near the berry then he grabs the berry and I don’t really get the ‘elsewhere.’”

Will noticed the later introduction of the audio features and thought they helped him. His favorite lesson was lesson 24. It was “not too hard and didn’t have as much going on.” He least liked lesson 23 because it was “way too much.” A feature Will wished was available in KW was the option to choose the main character and go through the lessons with it instead of always having to use Kokopelli.

Will did use the block reading audio feature, but not to help understand his code. He thought the way the text-to-speech pronounced “Kokopelli” was amusing and played it several times. Will thought the “Say My Code” option would be useful but did not use it himself because his program “kinda just read itself in my mind.” He described a hypothetical scenario in which it might be useful where he had made a larger program and then left it alone for a day or two before returning to it. He thought the audio feature would be useful for refreshing his understanding of his own code. He also thought the feature would have been helpful in his time with Scratch.

Will thought his time in KW more resembled reading and writing than doing math. Specifically lesson 24, “the maze one”, also reminded him of playing video games. Like with



Scratch, Will did not think profession programmers would use a system like Kokopelli's World either.

## 6. Discussion

This chapter discusses the implications of the results described in the previous chapter and is divided according to the separate Kokopelli's World versions that were studied.

### 6.1 Reflections on the KW HAWK Results

The KW HAWK intervention was our first test of KW with students and highlighted most of the important questions about dyslexic students working with blocks-based programming environments. Dyslexic students were able to learn coding techniques in our particular environment and, from questionnaires, we know they had successful coding experiences elsewhere too. This established a baseline for us to work from. Students also enjoyed the coding activities, they often preferred them to reading and writing activities. We expect that if dyslexics had as much trouble coding as they did reading they would not have felt better about coding. This somewhat corroborates the results from Powell et al. who found that dyslexics may be well-suited to some programming roles. That study theorized that the “recognition of need”, “problem definition” and “synthesis” steps involved in program generation would benefit from features of dyslexia that are typically considered positive.

Responses to the type of programming students preferred were almost evenly divided. Nearly half of the students preferred a more guided, puzzle-like series of lessons while the other half preferred more open-ended opportunities to express their own creativity. This is not a new result but shows that the different interests and preferences shown by typically developing students carry over to those with dyslexia. The KW Camp intervention was a first step to adapt KW to cater more to the creatively-minded students, and KW Class included two more open options even with its smaller allotted time.

Quantitatively, The KW HAWK research data showed some promise. We saw some correlation between reading/writing measures and success in KW. In so many words, children who

did well on certain reading and writing measures that were predictors of dyslexia also did well in the programming tasks. Since children who perform poorly in these measures are more likely to be diagnosed as dyslexic, this implies dyslexics may also be more likely to have difficulty with programming, at least in KW. There could be many explanations for these results, however, but the results go some way towards defining the relationship between dyslexia and programming.

One aspect clearly missing from the KW HAWK intervention was participation of typically developing students. Without data from them we had no way of establishing any kind of difference between dyslexic and typical student experiences in HAWK. This limitation was not present in the KW Class intervention.

## 6.2 Reflections on the KW Camp Results

Detailed in the Kokopelli's World design chapter, the KW Camp intervention was not formally conducted as a complete, separate experiment, but did provide some insight into the situations in which KW may or may not work well. The venue for KW Camp was an open-format summer coding camp for late-elementary school students. Sessions lasted 1-1.5 hours and had themes but no strict curriculum. Students were instead encouraged to engage with two coding environments we had made available to them, one of which was KW. The other system was BlockStudio, an environment designed with this type of open format in mind, which had a higher ceiling in the types of programs it could create and was more explicitly about creating games rather than stories (Banerjee et al. 2018). We suspect these are the reasons the children were more willing to engage with it than KW. Most of the students in the camp also had some programming experience already, which we did not expect. One of the camp themes was on transfer, and we had hoped to teach programming concepts through blocks in KW and programming by example in the other system. In most instances concepts would be explicitly taught in KW and then explored

further in BlockStudio. Since many students entered the class with some programming knowledge already this theme may not have been of much interest to them.

This experience showed us the current upper limits of KW Camp. In an open format and placed against more specialized or commercial systems, KW Camp did not seem as attractive to the camp population. It also does not have as much computational expressive power to offer students with programming experience already. We carried these lessons over into our design of KW Class, and this is further discussed in the future work section.

### 6.3 Reflections on the KW Class Results

Shifting focus to the KW Class intervention, we received a wider variety of results than in our previous studies. From the questionnaires we know students continued to feel mostly positive about their experiences with KW, though the feeling appears more muted than in the first study. This may be because there were more commercial or online options to compare it to. In KW HAWK the students only had other HAWK lessons to compare KW to, but in the classroom students had a larger variety of things they could do, including some commercial software, once they finished the lessons. From KW Camp we already knew KW would have difficulty competing for attention with commercially developed systems. We also know that all students in the KW Class study had at least some experience with professionally developed programming environments before, such as Hour of Code offered through Code.org. This was true for only some of the participants in KW HAWK.

Also from the questionnaires, we know students at least noticed the presence of the audio support and thought they were using it on average a few times per lesson. Students seemed to attribute that to the audio instructions before each lesson though, since responses did not noticeably differ between the experimental and control groups with respect to audio usage.

When we asked them about the connection between programming and either storytelling or games, students were able to draw connections to both. The majority of students saw some

similarities between programming and either making or experiencing games and stories. The storytelling results support the connection between coding and natural language literacy that was first explored in KW HAWK. This was found even without the emphasis on storytelling present in the HAWK reading and writing instructional materials, which were not presented in the KW Class intervention. This connection is something that warrants further exploration in future studies.

From the responses gathered while interviewing two particular students from the KW Class intervention we have further insight into what some dyslexic students feel about programming and the unique features of KW. Both of these students had previously done programming, and were interested in doing more, but both mentioned reading as a potential hindrance to one or more parts of the programming process. One student, Will, described code review as an instance where the KW Class' static reading feature might be useful. The other student Katy's response suggested that a system with more complex operations, or one without a microworld, could benefit more from a feature like the execution reading. Both of these students felt confident in their understanding of the blocks in KW but, matching the overall performance of KW Class, they were not able to describe the function of the conditional block with clarity.

In the results chapter we presented a set of block and behavior patterns that emerged while analyzing student work and took a detailed look at a few students' experiences through the whole body of lessons. Through this we saw clear instances of students actively learning while interacting with KW, and some common misconceptions. We also saw these misconceptions occur in both the KW HAWK and KW Class instances suggesting they were not unique to one or the other version.

The block patterns related to control structures were the most problematical for students. Attempts at three loop patterns were judged incorrect in most circumstances. *Ignore-the-loop* and *ignore-the-if* instances suggest students in those cases were either uninterested in following directions or failed to understand them. The *Wrap-the-loop*, *wrap-the-function*, *shuffle-the-if*, and *lose-the-loop* patterns indicate some students were failing to understand the intended use of loops

and functions but were at least trying to incorporate them. These patterns can inform how we might improve the lesson instructions or design.

*Early-discovery* and *special-event* were the two patterns we considered to be indicators of student success. With the limited block palette in each lesson we find it impressive when students manage to demonstrate deeper understanding. Most of the *early-discovery* instances occurred in early lessons when students discovered they could stack commands before being told as much. The other instances occurred when students nested control structures together before being told that they could. Unfortunately, we have no way of knowing if these students were drawing on past experience with other blocks languages or truly making new discoveries but these instances were still encouraging to see. The *special-event* pattern happened less frequently and with less clarity but, when it did occur, suggested a level of forethought we didn't expect of most students.

The other patterns we considered to be largely neutral in terms of student success. The *deja-fait* pattern was a surprisingly common occurrence that does not necessarily show misunderstanding on the students' part but may suggest uncertainty or a lack of attention. The *sequential-build* pattern is a viable coding strategy and did not co-occur with unsuccessful code but could possibly relate to working memory issues which dyslexics can be known to demonstrate. The *every-other-line* pattern only was demonstrated once but was noticeable enough we considered it worth mentioning. It could be the student adopted this pattern for aesthetic reasons, but it is also possible this related to his or her dyslexic condition. Dyslexics are known to be disproportionately affected by text size and spacing. It is possible the student structured the code this way to make it easier to read.

The *backwards-build* pattern tested design decisions that we had taken for granted initially. We had intended for most if not all commands in KW to be constructed in reading order, but the prevalence of this pattern showed that was not the preference for some students. This pattern occurred most often early in the set of lessons, before the introduction of control blocks. Due to the way KW is structured, it is more difficult to connect commands together without building them subject-first, since commands only connect to other commands via the subject block. Building

commands verb-first instead requires extra actions. We saw students build commands in reverse order less frequently in later lessons but it is not clear if they had altered their way of thinking or were simply taking the path of least resistance.

One reason why students might prefer to create commands verb-first is because KW initially contains only one microworld character: Kokopelli himself. At the point multi-part commands are introduced students may see the Kokopelli subject block as redundant because there is no other character that could be specified instead. When the subject can be assumed, the more mentally taxing aspect of each command would naturally become the verb. This may change when lessons first introduce the second character: Raven. The English language does not often use a verb-first sentence structure so this may also indicate an incompatibility with how students perceive and create programming commands versus how they perceive and create English sentences.

Counter to the results of KW HAWK, in KW Class we did not see a statistically significant correlation between coding performance and the reading test score. The reading test used for KW Class was not part of the test battery used in the KW HAWK experiment. It is possible that the effect disappeared when we removed the reading and writing instruction and left only coding, or when the number of lessons was pared down, but we may also consider this as initial evidence that different aspects of dyslexia can affect programming differently. In either case, while we cannot definitively prove the null hypothesis, that performance on the reading test does not correlate with programming success, we consider the lack of statistically significant differences an encouraging result. The majority of dyslexia measures studied in KW HAWK and the single measure used in KW Class did not show significant correlations with any aspect of programming. This suggests that a student's success in learning to code is not significantly affected by his or her reading ability, at least up to a point. This is supported by prior related work that indicated dyslexics may even be well-suited to programming, rather than disadvantaged. More work will be needed to further explore the topic, but this should be taken as evidence that dyslexia is not an insurmountable barrier for children learning to code.

## 6.4 Difficulty of Testing Target Population

Our experiments have shown some of the challenges for future studies to measure programming data from dyslexics with any statistical significance. Definitively diagnosed dyslexics only make up a small percentage of any random population and at this time it would appear any differences in coding performance they exhibit vis-à-vis non-dyslexics are small. Recruiting subjects from a small population for in-person experiments is resource intensive. We had considered conducting KW experiments online, but without access to a reliable, online test for dyslexia we were reluctant to trust the unvetted word of any participants. This issue is further discussed in the future work chapter, along with a potential solution.

As programming has become more popular among younger people, it has also become increasingly difficult to find children with no coding experience, confounding measures of programming growth with a particular system. Others who are working on language-independent learning assessment methods may make this easier (Nelson et al. 2019).

## 6.5 Research Hypotheses

Our work has informed directions for future research into the relationship between dyslexia and children's programming. This section lists several research hypotheses we believe will prove fruitful based on the work we have done here.

- Each distinct reading, writing and cognitive measure associated with dyslexia will exhibit a different correlation with success in programming activities.
- The correlations measured as part of the hypothesis above will be a mixture of both positive and negative with respect to coding success. i.e. dyslexic students will outperform typically developing students in some measures of coding success and underperform in others.
- Usage rate of the coding patterns introduced in this thesis will also show different values of correlation with the various dyslexia measures.



- The same coding patterns will also appropriately represent programming behavior of typically developing students
- Audio support features in a blocks coding environment will become more desired and used by students as program complexity increases. This effect will be more pronounced for dyslexic than for typically developing students.
- Switching from a text-based programming environment to a blocks-based one will cause a greater reduction in programming errors by dyslexic students than by typically developing ones.

Suggestions for how these might be tested are shared in the future work chapter.

## 7. Future Work

This research presents many possible directions for future work. We have divided up suggestions into future possible work for KW, suggestions for others studying learning disabilities and coding, and some high-level thoughts on coding and literacy education for children.

### 7.1 Future Changes to Kokopelli's World

Starting at the smallest scale of future changes, even after design iteration we know that conditionals remain the most difficult aspect of KW programming for students to understand. Further work should be done to improve those particular lessons. This might involve reducing the ambiguity of language in the instructions and blocks, or including new lessons to eliminate more misconceptions, or altering the KW language itself to allow for more specific use cases where programs benefit from conditional control structures.

Some students expressed appreciation for the challenge of later lessons in KW that used conditionals, but from anecdotal responses the most popular lesson in KW Class was lesson 24, which gave students a specific goal but left them with more freedom for how to achieve it than other lessons. Refining our code checking methods and offering more of these types of problems in a future version could increase student success.

One factor that severely inhibits the scale-up of studies with programming and learning disabilities in general is the necessity to know if individuals are learning disabled or not. KW uses one measure that has been associated with dyslexia as the “reading test”, but dyslexia is a complicated condition that can manifest in multiple, orthogonal ways. Other learning disabilities also manifest in a variety of ways. Creating or incorporating a comprehensive series of diagnostic measures in KW that could determine the user's condition accurately but remotely would either

eliminate the need to collect that information offline from parents or at least supplement such information. Others are currently working on this (Rauschenberger 2018). If such a system became fully online then it could be run on a much larger scale than how it has been used up to this point. This could also be used as an incentive to attract participants looking to know if they might have a learning disability.

Such a system could also be leveraged to perform more detailed tests examining the differences between dyslexic and typically developing students. With a large enough number of subjects even small effects could be discovered.

Currently KW is designed to be completed in a small number of sittings. Users complete the pre-defined tasks, try one or two open-ended programs, and are then expected to be done. Creating an environment where users would want to regularly return to KW would greatly increase the data KW could gather and fostering a community around KW could create a self-sustaining user base. KW Camp represented some initial but insufficient efforts towards this goal. Future versions of KW could add community features akin to what Scratch provides, with the ability to comment on, copy, and remix other people's projects.

The skill ceiling of KW would also need to be increased to keep experienced users engaged for longer. This could include the eventual introduction of variables and a move away from the turn-based, single-threaded execution method KW currently uses. More flexible control structures could also help, with new features such as while loops, dynamic for loops, event handlers, and more if conditions.

This increased program complexity could also be used to further study KW's audio features. Both dyslexic students that were interviewed thought that the audio features would be more useful with more complex programs. A remixing or sharing feature would also create situations where students would need to read code created by other people, which could also be a situation where text-to-speech features might help.

Screen readers are one of the most popular tools used by dyslexics to assist in accessing material mediated by technology. While KW has recreated many of the features available in screen readers, and added several that aren't, there is still more than can be done on this front. KW Class offered the limited ability to adjust the audio playback speed, but this should be expanded in future versions. Another potential addition is the ability to choose the voice the computer uses to read aloud. A drop-down menu could be added to KW to let users choose the voice they wish to hear. One downside is KW relies on the native web browser text-to-speech library, which means the voices available depend on what browser is used. In some cases these choices may be very limited.

A strategy that has proven successful with LD students in other subjects, including our own HAWK system, is differentiated instruction. It was outside the scope of this research to create individual curricula for each student but integrating KW or a similar system into an intelligent tutoring system might see greater effect sizes as their individual needs are better met. This is doubly true as programming becomes more popular and new students enter the field potentially having already had a wide range of programming experiences. It is more likely than ever that students will develop at different rates that make teaching from one static curriculum quite hard.

The patterns observed in the dyslexic students' programs from the KW HAWK and KW Class experiments show some promise. The program data for only one typically developing student was coded this way, but the same treatment should be done for the other typically developing students. Ideally we would see the patterns still represented in some way. The coding for these patterns was done by hand, but ideally some or all of them could eventually be detected automatically, potentially through a machine learning method.

## 7.2 Learning Disabilities and Coding

Outside of the work done with HAWK and KW, we believe there are still many unanswered questions about the relationship between programming and learning disabilities.

One potentially valuable study would involve surveying the current professional programming community to determine how well learning disabled individuals are represented and if that proportion is different from the general population. If we have some notion of whether learning disabled people naturally gravitate to or away from programming that would be a key starting point. The same studies could also be done with undergraduate and graduate computer science programs in universities. If disproportionately large numbers of learning disabled individuals are coming to computer science then more work needs to be done to understand and accommodate these students and their particular needs. It may also be a sign that programming is even well-suited to people with these conditions, and they should be encouraged to consider programming when they are still students. On the other hand, if the learning-disabled portion is disproportionately low, then that means the computer science community has done a poor job of accommodating people in this category and should strive to do better. In either case, a study of this sort could set the tenor of future work looking into these conditions. The only attempt at this kind of study we are aware of asked a set of 315 Spanish-speaking programmers if they thought they were dyslexic and found 12.4% responded “yes” (Gonzalez 2016). This percentage is larger than some general population estimations and suggests dyslexia and programming may be compatible, but more work of this type will be needed for it to be conclusive.

Whether it be to accommodate dyslexics or vision-impaired students, more programming environments should strive to support text-to-speech software or incorporate their own audio. A survey by Milne (2018) showed that most popular blocks languages are not accessible to vision-impaired students, who sometimes use the same screen-readers dyslexics do. Particularly since blocks languages are being used to introduce children and other new programmers to the field, it becomes even more important that such systems be as accessible as possible to avoid discouraging novices.

This research has focused nearly exclusively on dyslexia. This is largely due to circumstance and how prevalent dyslexia is compared to other conditions. In the process of

conducting this research we have noticed little to no work on other learning disabled conditions and their relationship to programming. Dyslexia was not unique in that regard. Dyscalculia, for instance, may be a uniquely large hindrance for potential future programmers, as it affects mental calculation processes in particular. This warrants further study and - considering the wide variety of behaviors that develop out of the various learning disabilities - a system that addresses the needs of dyscalculics may look as different from KW as it does from traditional programming systems.

### 7.3 A Possible Model of Dyslexia-Programming Interaction

Strict code production is not the only aspect of software engineering as a field. While it is the first and primary aspect taught to children, professional programmers must draw upon many more skills than just creating code. Most of these tasks are outside the scope of this thesis, but a broad effort to fully accommodate dyslexics in the field of programming should consider them, particularly if we consider programming in terms of software engineering. We have created a matrix comparing skills and barriers associated with dyslexia, delineated in Powell et al.'s work, to activities that make up modern programming according to Alan Blackwell's "What is programming?" (Powell et al. 2004; Blackwell 2014). Blackwell mentions five broad types of activity: Requirements, Specification, Design, Coding, and Debugging. We mark negative interactions, where, we hypothesize, a dyslexic individual may have difficulties with an 'X' and positive interactions, where such an individual might benefit from one of his or her practiced skills, with an 'O'. This matrix can be seen in Table .

*Table 7.1: Matrix of possible interactions between the strengths and weaknesses of dyslexics and the skills involved in computer programming. Positive interactions are marked with circles, negative interactions are marked with x's.*

	Requirements	Specification	Design	Coding	Debugging
Handwriting					
Spelling	X			X	
Reading	X			X	X
Writing	X			X	
Short Term Memory		X	X	X	X
Spatial Awareness					O
Visualization			O		O
Creativity		O	O		
Lateral Thinking		O	O		

Requirements are typically the desired results of executing a program, and the process of deciding what those should be. We consider this a largely social activity, when separated from design, that would include a lot of written, asynchronous communication that could hinder dyslexics. Fortunately these sorts of activities are not unique to programming and tools from other disciplines can be utilized.

Specification is the process of defining the context and circumstances in which a program will be used after the initial requirements have been established. We see this as a phase of program creation where dyslexics could benefit from their strong creative and lateral thinking skills.

Design is often considered a complex problem-solving process and can include a variety of activities (Simon 1988). The key stages of design are more often mental or visual than based in text,

however, so we consider handwriting, spelling, reading, and writing skills to be non-critical at this stage. Short-term memory is applicable since design is certainly a mental process, and visualization, creativity, and lateral thinking are relevant for similar reasons.

Program implementation, or coding, can be considered a simpler task when separated from design. We have already shown that actually writing code is possible for dyslexic children using a blocks language, but modern IDE affordances like code completion may also do much to offset difficulties with spelling and writing. Considering how a code base is organized and navigated may rely on spatial awareness skills.

Software debugging may also be affected by short-term memory limitations, since it can involve complex mental models of program execution. Debugging is also more likely to involve reading code, especially written by other people, which could hinder dyslexics. Just as with coding, spatial awareness would help code space navigation, and visualization would be useful for communicating how an error is occurring.

The documentation of software often involves the production of a large amount of text, and this is where dyslexics are most likely to have difficulties. Reading, writing, and spelling issues could make documentation more laborious. Dyslexics will also have fewer advantages to draw on, except potentially using visuals to convey aspects of program architecture.

A study that measures these aspects of dyslexia, positive and negative, could find correlation with aspects of programming beyond just straightforward coding. In particular, this could be useful for dyslexic adults planning to enter the world of professional programming.

## 7.4 Coding and Literacy

The KW HAWK study hinted at a potential connection between natural language literacy and coding literacy. Survey results from KW Class also show a majority of students were able to conceive of coding as a form of storytelling, even with very little structure to encourage that



comparison. We believe there is potential benefit from further exploring this connection. One can consider the possibility that if coding, using a near-natural-programming-language, were taught at the same time as writing, coding could be used as an automatic feedback method to critique spelling or simple grammar structures. Code can be parsed and executed as well, which could provide a strong incentive for students to engage with it. The KW HAWK study showed that coding was a strong motivator for some students. If this were re-creatable in other writing contexts it could be used to encourage students who view writing as an otherwise dry topic. Rather than just seeing their words on a page or screen, they could see them brought to life in a microworld or some other type of programmatically produced multimedia. As handwriting instruction slowly disappears from standard curricula, coding can become a natural extension of typing. With this approach, potentially even more people could receive basic coding instruction than are currently and in a way naturally integrated with traditional education topics.

## 8. Conclusion

Computer Science as a field benefits when groups from all walks of life are able to make contributions inspired by their own unique experiences and opinions. The group of individuals with learning disabilities, dyslexics included, is no exception.

The condition of dyslexia is still being studied and has yet to be fully understood. It is known to affect reading, writing, and spelling among other skills, but computerized educational systems like HAWK show that dyslexic students do not have to be expected to perform worse than their peers in these topics.

Far less is known about the relationship between programming and dyslexia. While programming can involve forms of reading, writing, and spelling, it draws upon many other skills as well. Kokopelli's World and HAWK are some of the first probes into exploring these interactions. We have shown not only that dyslexic children are capable of and enthusiastic about learning to program, but also that there are potential interactions between dyslexia measures and coding ability that we do not yet fully understand. Different measures associated with dyslexia may have different interactions with programming measures. The "word choice" measure showed some positive correlation with the ability to complete coding lessons quickly, but we did not see a similar effect from the other dyslexia measures, including the reading test used as part of KW Class. More studies exploring these topics should be performed.

This thesis also presented a set of block-based programming and behavior patterns observed among the dyslexic students participating in KW studies. These patterns were shown to differentiate students and in some cases show their current understanding of key programming concepts such as loops and conditionals.

As programming grows as a field and profession, and as dyslexia diagnoses grow at the same time, these results and this topic in general warrant further study to ensure that programming remains accessible to all who are interested in it.

## Works Cited

- R. Abbott, M. Raskind, M. Matsushita, R. Richards, N. Price, & V. Berninger. (2017). Dysgraphia, dyslexia, and OWL LD during middle childhood and early adolescence: Evidence for genetic effects on hallmark phenotypes. *Biomarkers and Genes*, 1 (1), 1-10. doi: 10.15761/BG.1000103 NIHMSID 950942
- Acorn. (2019). A small, fast Javascript-based Javascript parser. URL: <https://github.com/acornjs/acorn>.
- H. Abelson, N. Goodman & L. Rudolph. (1974). LOGO Manual. Artificial Intelligence Lab, Massachusetts Institute of Technology. December 1, 1974. URI: <http://hdl.handle.net/1721.1/6226>.
- R. Banerjee, L. Liu, K. Sobel, C. Pitt, K.J. Lee, M. Wang, S. Chen, L. Davison, J. Yip, A. Ko, & Z. Popović. (2018). Empowering families facing English literacy challenges to jointly engage in computer programming. In *Proceedings of SIGCHI Human Factors in Computing Systems (CHI 2018)*.
- D. Bear, M. Iverezzi, S. Templeton, & F. Johnston. (2000). *Words their way: Word study for phonics, vocabulary, and spelling instruction* ( 2nd ed.). NJ: Merrill: Upper Saddle River.
- V. Berninger, W. Nagy, S. Tanimoto, R. Thompson, & R. Abbott. (2015, published on line October 30, 2014). Computer instruction in handwriting, spelling, and composing for students with specific learning disabilities in grades 4 to 9. *Computers and Education*, 81, 154-168. DOI information: 10.1016/j.compedu.2014.10.00NIHMS636683
- V. Berninger & T. Richards. (2010). Inter-relationships among behavioral markers, genes, brain, and treatment in dyslexia and dysgraphia. *Future Neurology*. 2010;5:597–617. doi: 10.2217/fnl.10.22. NIHMS 226931.
- V. Berninger, T. Richards, & R. Abbott. (2015, published on line April 21, 2015). Differential diagnosis of dysgraphia, dyslexia, and OWL LD: Behavioral and neuroimaging evidence. *Reading and Writing. An Interdisciplinary Journal*, 28, 1119-1153. doi:10.1007/s11145-015-9565-0 A2 contains supplementary material available to authorized users: NIHMS683238 Publ ID 2615-04-21\_0002 Released to PMCID 4553247
- V. Berninger, W. Winn, P. Stock, R. Abbott, K. Eschen, C. Lin, N. Garcia, M. Anderson-Youngstrom, H. Murphy, D. Lovitt, P. Trivedi, J. Jones, D. Amtmann, & W. Nagy. (2008). Tier 3 specialized writing instruction for students with dyslexia. *Reading and Writing. An Interdisciplinary Journal*, 21, 95-129. Printed Springer On Line. May 15, 2007.
- A. Blackwell. (2014) What is programming? *Psychology of Programming Interest Group*, 204-218.
- M. Bostock. (2019) D3 Data Driven Documents. URL: <https://d3js.org/>.
- M. Brault. (2012). *Americans with disabilities: 2010* (pp. 1-23). Washington, DC: US Department of Commerce, Economics and Statistics Administration, US Census Bureau.
- B. Butterworth & Y. Kovas. (2013). Understanding Neurocognitive Developmental Disorders Can Improve Education for All. *Science*, 2013; 340 (6130): 300 DOI: 10.1126/science.1231022

- J. L. Castro, L. F. Alvaran & L. A. M. Normand. (2018). Visual Programming Languages for Programmers with Dyslexia: An Experiment. In IEEE e-Science, 2018; Amsterdam, Netherlands. DOI: 10.1109/eScience.2018.00030.
- M. Conway, R. Pausch, R. Gossweiler & T. Burnette. (1994) Alice: A rapid prototyping system for building virtual environments. In *Conference Companion on Human Factors in Computing Systems* (CHI '94), Catherine Plaisant (Ed.). ACM, New York, NY, USA, 295-296.
- Code.org. (2014) Research at Code.org. URL: <https://code.org/research>.
- S. Dasgupta, B. M. Hill. (2017) Measuring learning of code patterns in informal learning environments. SIGCSE 2017, 706-706. DOI: 10.1145/3017680.3022437.
- Department of Education. (2010). Twenty-ninth annual report to congress on the implementation of the Individuals with Disabilities Education Act, Parts B and C. 2007. Retrieved June 12, 2012, from <http://www2.ed.gov/about/reports/annual/osep/2007/parts-b-c/>.
- E. A. Draffan, Dr D. G. Evans & P. Blenkhorn. (2007). Use of assistive technology by students with dyslexia in post-secondary education. *Disability and Rehabilitation: Assistive Technology*, 2:2, 105-116, DOI: 10.1080/17483100601178492.
- EllisLab. (2019) CodeIgniter. URL: <https://codeigniter.com/>.
- L. P. Flannery, B. Silverman, E. R. Kazakoff, M. U. Bers, P. Bonta & M. Resnick. (2013) Designing ScratchJr: Support for early childhood learning through computer programming. IDC '13 Proceedings of the 12th International Conference on Interaction Design and Children, 1-10.
- M. Fowler. (1999) Refactoring: improving the design of existing code. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.
- N. Fraser. (2013). Blockly: A visual programming editor. URL: <https://code.google.com/p/blockly>.
- B. Gardiner. (2014, March 23). Adding coding to the curriculum. The New York Times. <https://www.nytimes.com/2014/03/24/world/europe/adding-coding-to-the-curriculum.html> 2/5/18. Accessed Sep 19, 2018.
- L. Gonzalez. (2016) Encuesta tesis doctoral Transformación bidireccional de código de software visual y texto para mitigar los efectos de la dislexia en el desarrollo de software. [online] Available: <http://oa.upm.es/39155/>.
- R. B. Gottier. (2019) Community Auth for Code Igniter 3. URL: <https://community-auth.com/>.
- L. Gould & W. Finzer (1984) Programming by Rehearsal. Xerox Corp. Palo Alto Res. Center Report No. SCL-84-1, May 1984.
- J. Gray, H. Abelson, D. Wolber & M. Friend. (2012). Teaching CS principles with app inventor. In Proceedings of the 50th Annual Southeast Regional Conference ACM, 405-406.
- N. Halfon, A. Houtrow, K. Larson & PW. Newacheck. (2012). The changing landscape of disability in childhood. *Future of Children*. 2012;22(1):13-42.

- T. Hall, C. Hughes & M. Filbert. (2000). Computer-assisted instruction in reading for students with learning disabilities: a research synthesis. *Education and Treatment of Children*, 23:2 (May 2000) 173-193.
- L. Hardesty. (2010). The MIT roots of Google's new software. MIT News Office. August 19, 2010.
- M. Henry. (2010). *Unlocking literacy. Effective decoding and spelling instruction* ( 2nd ed.). Baltimore: Paul H. Brookes Publishing.
- F. Hermans & E. Aivaloglou. (2016, May 16) Do code smells hamper novice programming? A controlled experiment on Scratch programs. In 2016 IEEE 24th International Conference on Program Comprehension (ICPC). DOI: 10.1109/ICPC.2016.7503706.
- K. Howland & J. Good. (2015) Learning to communicate computationally with Flip: A bi-modal programming language for game creation. *Computers & Education*, 80, 224-240.
- InfoWorld. (1983). Software Reviews: Spinnaker. InfoWorld Vol. 6, No. 33, 56-58.
- International Dyslexia Association. (2017). Attention-Deficit/Hyperactivity Disorder (AD/HD) and Dyslexia. <https://dyslexiaida.org/attention-deficithyperactivity-disorder-adhd-and-dyslexia/> Accessed Sep. 18, 2018.
- K. Kahn. (1996) ToonTalk™ – An animated programming environment for children. *The Journal of Visual Languages and Computing*, April 16, 7(2), 1-23.
- C. Kelleher & R. Pausch. (2007). Using storytelling to motivate programming. *Communications of the ACM*, July 2007, 50:7, 59-64.
- J. Kemeny & T. Kurtz. (1964). Basic: a manual for BASIC, the elementary algebraic language designed for use with the Dartmouth Time Sharing System (PDF) (1st ed.). Hanover, N.H.: Dartmouth College Computation Center.
- D. Kurlander, T. Skelly & D. Salesin. (1996). Comic Chat. *Proceeding of SIGGRAPH '96*. August 4-9, 1996, 225-236.
- M. Lee & A. Ko. (2011). Personifying programming tool feedback improves novice programmers' learning. In *Proceedings of the seventh international workshop on Computing education research* ACM, 109-116.
- H. Liu & H. Lieberman. (2005). Metafor: visualizing stories as code. In *Proceedings of the 10th international conference on Intelligent user interfaces* ACM, 305-307.
- N. Mather, R. Roberts, D. Hammill & E. Allen. (2008). *Test of Orthographic Competence (TOC)*. Austin, TX: Pro-Ed.
- I. McChesney & R. Bond. (2018). Eye tracking analysis of computer program comprehension in programmers with dyslexia. *Empirical Software Engineering*. June 2019, 24:3, 1109-1154.
- T.R. Miles, M.N. Haslum & T.J. Wheeler. (1998). *Ann. of Dyslexia*. 48: 27. <https://doi.org/10.1007/s11881-998-0003-8>

- L. Milne. (2018). Touchscreen-based learning technologies for children with visual impairments. (Doctoral Thesis). University of Washington, Seattle.
- L. Morgado, M. Cruz & K. Kahn. (2006) Radia Perlman – A pioneer of young children computer programming. *Current Developments in Technology-Assisted Education*, 1903-1908.
- G.L. Nelson, A. Hu, B. Xie, A. Ko. (2019) Towards validity for a formative assessment for language-specific program tracing skills. Koli Calling, '19 November 21-24, 2019, Koli, Finland.
- J. Niedo, S. Tanimoto, R. Thompson, R. Abbott & V. Berninger. (2016). Computerized instruction in translation strategies for students in upper elementary and middle school grades with persisting learning disabilities in written language. *Learning Disabilities. A Multidisciplinary Journal*, 21, 62-78. NIHMS 836952
- E. Norton, S. Beach & J. Gabrieli. (2014). Neurobiology of dyslexia. *Curr Opin Neurobiol*. 2015 Feb; 0: 73–78.
- S. Papert. (1980) *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books. New York.
- S. Papert & C. Solomon. (1972) Twenty things to do with a computer. MIT, AI Lab. *LOGO Memo* 3, July 1971. Also in *Educational Technology*, April 1972.
- Pearson (2009). Wechsler Individual Achievement Test, 3rd Ed. San Antonio, TX.
- N. Powell, D. Moore, J. Gray, J. Finlay & J. Reaney. (2004). Dyslexia and learning computer programming.
- M. Rauschenberger, L. Rello, R. Baeza-Yates & J. P. Bigham. (2018). Towards Language Independent Detection of Dyslexia with a Web-based Game. W4A 2018: The 18th Conference on Web Accessibility. Lyon, France, ACM, New York, NY, USA.
- A. Repenning. (1991). Creating user interfaces with agentsheets. In 1991 IEEE Symposium on Applied Computing, 190-196.
- M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman & Y. Kafai. (2009) Scratch: Programming for all. *Communications of the ACM*, 52:11 (November 2009), 60-67.
- T. Richards, S. Peverly, A. Wolf, R. Abbott, S. Tanimoto, R. Thompson, W. Nagy & V. Berninger. (2016, October 1; July 22 on line). Idea units in notes and summaries for read texts by keyboard and pencil in middle childhood students with specific learning disabilities: Cognitive and brain findings. *Trends in Neuroscience and Education*., 5, 146-155.
- J. Sammet. (1966, March). The use of English as a programming language. *Communications of the ACM*, 9:3, 228-230.
- Y. Seo & D. Bryant. (2009) Analysis of studies of the effects of computer-assisted instruction on the mathematics performance of students with learning disabilities. *Computers and Education*, 53 (May 4, 2009), 913-928.

- S. Shaywitz & B. Shaywitz. (2005). Dyslexia (specific reading disability). *Biological psychiatry*, 57(11), 1301-1309.
- L. S. Siegel. (2006). Perspectives on dyslexia. *Paediatrics & child health*, 11(9), 581–587. doi:10.1093/pch/11.9.581.
- J. Siegmund, C. Castner, S. Apel, C. Parnin, A. Bethmann, T. Leich, G. Saake & A. Brechmann. (2014) Understanding understanding source code with functional magnetic resonance imaging. *ICSE 2014*, (May 31, 2014), 378-389.
- H. Simon. (1988). The science of design: Creating the artificial. *Design Issues*, 67-82.
- J. Simpson. (2019). Howler.js audio library for the modern web. URL: <https://howlerjs.com/>.
- M. Singson, D. Mahony & V. Mann. (2000). The relation between reading ability and morphological skills: Evidence from derivational suffixes. *Reading and Writing: An Interdisciplinary Journal*, 12, 219– 252.
- I. Sutherland. (1963). Sketchpad: A man-machine graphical communication system. courtesy Computer Laboratory, University of Cambridge UCAM-CL-TR-574 September 2003. Massachusetts Institute of Technology.
- S. L. Tanimoto & M. S. Runyan (1986) PLAY: An iconic programming system for children. In S.-K. Chang, T. Ichikawa, and P. A. Ligomenides (eds.) *Visual Languages*. New York: Plenum Press, 191-205.
- S. Tanimoto, R. Thompson, V. Berninger, W. Nagy & R. Abbott. (2015). Computerized writing and reading instruction for students in grades 4 to 9 with specific learning disabilities affecting written language. *Journal of Computer Assisted Learning*, 31, 671-689. NIHMS 721216 \_doi: 10.1111/jcal.12110
- R. Thompson, S. Tanimoto, R. Abbott, K. Nielsen, K. Geselowitz, R. Lyman, K. Habermann, T. Mickail, M. Raskind, S. Peverly, W. Nagy & V. Berninger. (2016, July 19, on line). Relationships between language input and letter output modes in writing notes and summaries for students in grades 4 to 9 with persisting writing disabilities. *Assistive Technology Journal*. DOI: 10.1080/10400435.2016.1199066
- R. Thompson, S. Tanimoto, V. Berninger & W. Nagy. (2015). Design Studies for Stylus and Finger-Based Interaction in Writing Instruction on Tablets. WIPTTE 2015.
- R. Thompson, S. Tanimoto, R. Lyman, K. Geselowitz, K. Begay, K. Nielsen, R. Abbott, W. Nagy, M. Raskind & V. Berninger. (2017, September 17). Effective instruction for persisting dyslexia in upper grades: Adding hope stories and computer coding to explicit literacy instruction. *Education and Information Technology*.
- D. Thornburg. (1983). Friends of the Turtle: On Logo And Turtles. *Compute!*. p. 148.
- J. K. Torgesen, R. K. Wagner, & C. A. Rashotte. (1999). TOWRE Test of Word Reading Efficiency. Austin, TX: Pro-Ed.

G. Troia (Ed.). (2009). *Instruction and assessment for struggling writers. Evidence-based practices*. New York: Guilford.

United States National Institute of Child Health and Development. Learning Disabilities. Sep. 11, 2018. <https://www.nichd.nih.gov/health/topics/learningdisabilities>. Accessed Sep 17, 2018.

Wikipedia: The Free Encyclopedia. (2019). Kokopelli. 22 Sep. 2019. Retrieved 28 Oct. 2019. URL: <https://en.wikipedia.org/wiki/Kokopelli>

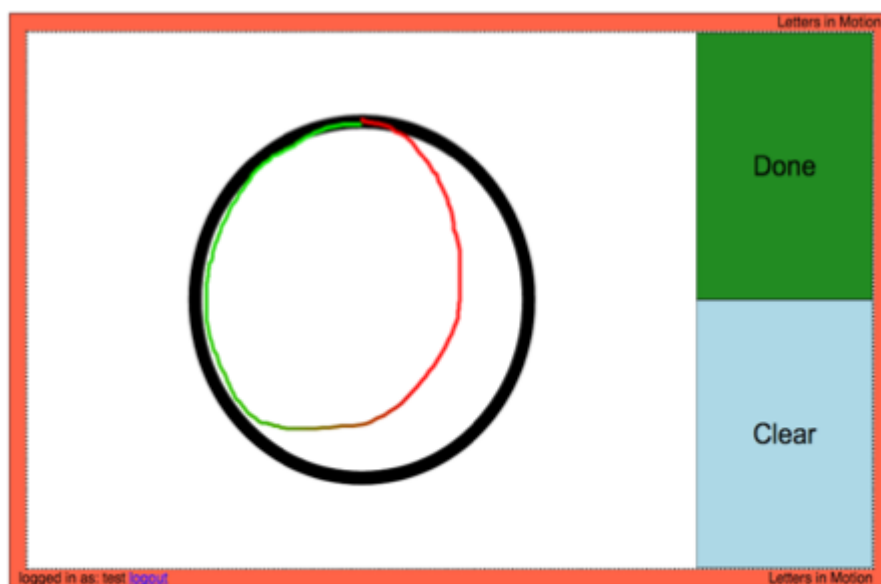
R. Woodcock, K. McGrew, & N. Mather. (2001). *Woodcock-Johnson III cognitive-achievement battery*. Itasca, IL: Riverside.



## Appendix A - HAWK Writing Feedback Methods

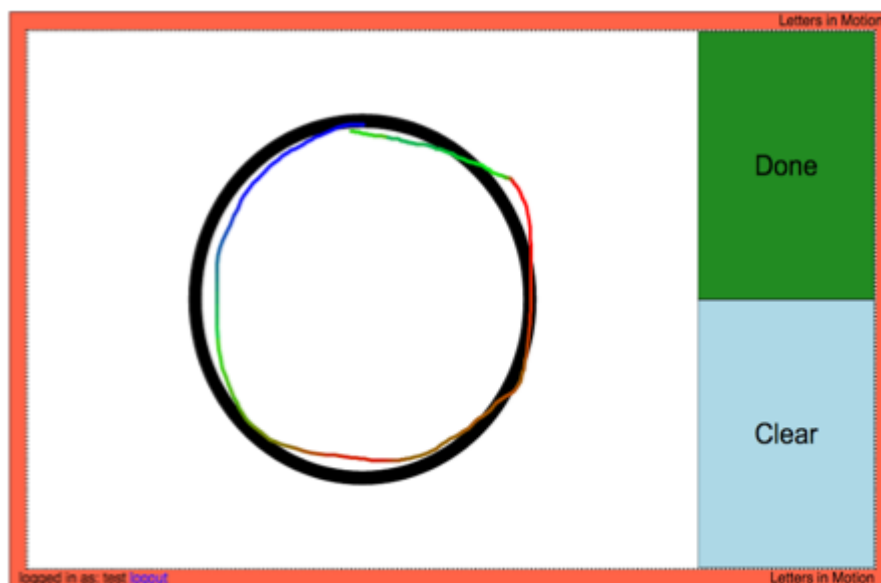
Handwriting recognition systems were available when HAWK was developed. However, unlike a lot of these systems the HAWK team was primarily interested in assessing the quality of handwriting, rather than only recognizing it. As such we developed several original, live writing assessment methods for use in the “Letters in Motion” section of HAWK devoted to practicing writing individual letters (Thompson 2015).

The first was the “length” method that, on a stroke by stroke basis, compared the length of a student’s stroke to a predetermined “ideal” stroke. Based on the current stroke length, a distance was calculated from the student’s pen’s position to where their pen should be following the ideal stroke. This number determined how well the letter was being written and could be used to provide feedback. In our case, the distance was used to color the student’s stroke as it was being written. A “good” stroke would be colored green, while a poor stroke would gradually shift to red as the student moved further and further from the ideal. An example of this feedback method for drawing the letter “O” can be seen in Figure 0.1.



*Figure 0.1: Writing feedback for a drawn letter ‘O’ using the length method.*

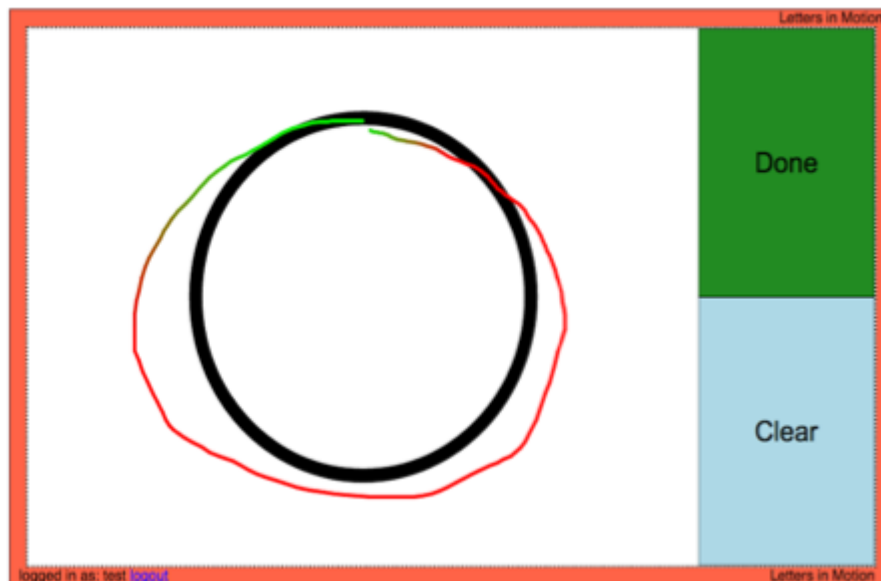
The second handwriting assessment method devised was the “time” method. In this case the student would be expected to follow a guiding pen-point that would move around the screen and trace the shape of the letter the student was expected to write. If a student was writing too slowly and “behind” the stroke then their stroke color would shift to blue to indicate they should speed up. If they moved too far ahead of the guide marker then their stroke color would shift to red to indicate they should slow down. In both cases the assessment was chiefly concerned with the distance from the student’s pen tip to the guide marker at a given moment in time. A perfectly-traced letter would still be incorrect if it was drawn too slowly.



*Figure 0.2: Writing feedback for a drawn letter ‘O’ using the time method.*

We also prototyped a “length+time” method that colored the user’s stroke using an average of the two methods described above. This prevented a stroke from being treated as mostly correct if a user got off-course and then corrected afterwards, which was allowed to happen in the pure time-based method. Example output for this method can be seen in Figure 0.3. All three of these methods were implemented but ultimately were deemed more precise than necessary for use with our audience of elementary-aged children. At a spatial scale that resembled regular handwriting, the color feedback on the screen was difficult to visually parse. Also, at a time scale that resembled

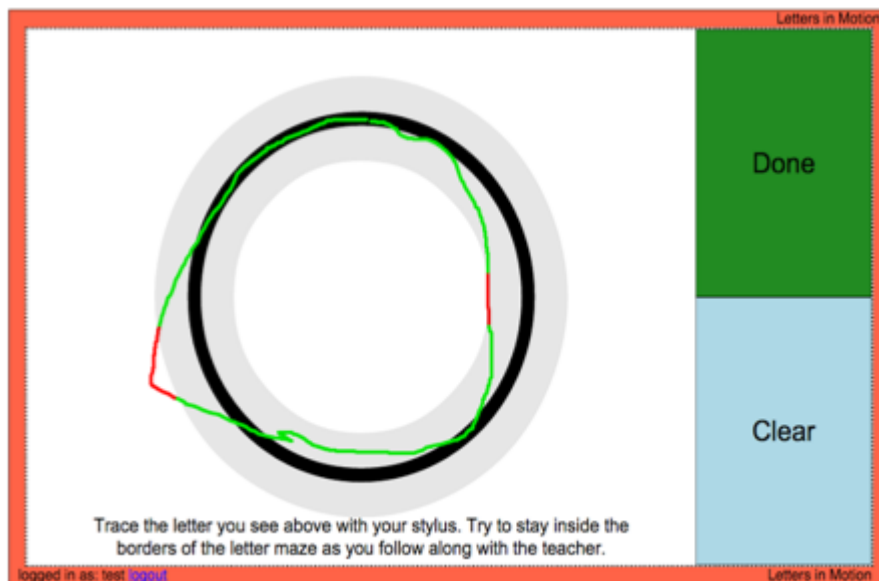
regular handwriting, there was little time to react to stroke feedback before the stroke was finished. These methods ended up more appropriate as feedback after-the-fact than live during writing.



*Figure 0.3: Writing feedback for a drawn letter 'O' using the length+time method.*

The 4th assessment method developed, that was eventually used in HAWK, was the “ductus” method. Like the earlier methods, the system has a predetermined model for what an “ideal” version of the letter looks like. The stroke width for this model is expanded to create very thick letters. These letters are then used as boundaries while the student user writes. If their stroke stays in the current letter “ductus” then their stroke is considered correct. If their stroke leaves the ductus then it is colored red and this instance is marked as a mistake. At the end of the lesson, the student is shown the number of mistakes they made over the course of writing several letters. These ductuses, or “letter mazes” as we called them in the lessons, were created for print and cursive alphabets in upper and lower case. This method also relied on large letters, and after an initial trial it was determined that the large scale required to write legibly with a finger on the iPad was inappropriate for the age and skill level of our subjects, who were typically in the 4th-9th grades.

The finger-based writing was also dropped in later studies in favor of stylus, typing, and pencil-and-paper writing.



*Figure 0.4: Writing feedback for a drawn letter ‘O’ using the ductus method.*

When the HAWK lessons expanded from writing single letters to full words and sentences we did not employ the ductus feedback method. Like with later versions of HAWK, we determined the ductus method to scale up poorly for more complex, smaller writing that would be required for writing multiple words and sentences. This is also why we did not seriously consider incorporating handwriting into Kokopelli’s World during its design phase. Instead, for longer-form writing activities we assessed students on secondary metrics such as how many words they typed, how many pages of text they wrote, or how many times they got off task during the writing activities, typically recognized as a period of inactivity that lasted at least 2 seconds. This was the limit of what we developed as live feedback, though all writing data was analyzed off-line in more detail at a later time.

## **Appendix B - HAWK Implementation Details**

Audio in HAWK was particularly challenging for a long period of time. Like many multimedia applications, HAWK requires audio be played at precise times to match user actions. By default the iOS browser does not allow more than one audio file cached in memory, and no audio is allowed to be played unless triggered by a user action. In practice this meant that lesson introductions couldn't start until the user tapped the screen and, more importantly, audio data couldn't start loading until that tap either. Keeping sound effect as separate audio files also meant that when a user event attached to audio triggered, there was almost always a delay as the new sound file had to be downloaded from the server every time. Our first solution to this was to use audio sprites, where every sound necessary for a particular activity was stitched together into a single audio file. Then different sections of that file were played as needed. This was not an ideal solution, however. Combined with JavaScript's single-threaded implementation and the iPad's low memory this often caused extraneous audio to bleed over. The iPads also occasionally has issues loading the larger audio file at all. Eventually, advancements in HTML5 audio and updates to the iOS browser allowed us to start caching more than one file and most of these problems were eliminated. Audio-synced multimedia on a mobile device web browser is still no small accomplishment however, and native audio support is still poor.

One concern of the project was the side effects that came with granting internet access to a group of rowdy children, many of whom had ADHD. Our first approach was to use a "safe" browser on the iPads that allowed whitelisting. This failed when the browser was unable to support several dozen simultaneous users. Eventually students were allowed to use the uninhibited default browser, but in return students had to be regularly monitored to keep them on-task. This became easier when programming in KW was introduced and students became more motivated to complete earlier tasks and reach the coding portion at the end.

Some of HAWK's most important contributions are centered around writing. HAWK writing data is recorded and stored as a series of points with x, y, and time coordinates. These points are grouped into lists by the stroke each point is associated with. Strokes can be further gathered into collections of strokes associated with full letters for the Letters in Motion portion. Strokes made during longer composition activities like in Minds in Motion are not separated by letter. Writing points are gathered as part of the mouseMove web event listener, and so writing data resolution is dependent on how quickly the JavaScript process can run and trigger new events. On older iPads with other processes competing for memory this can cause writing to appear "jaggy" when not enough points are able to be drawn in a short time span.

For letter writing feedback methods, HAWK also stores a set of SVG paths that represent the ideal path the student's writing should take to form a particular letter. HAWK has ideal paths for upper and lowercase versions of print and cursive alphabets. These paths can easily undergo transformations like rescaling or rotation if need be. SVG paths can also be easily animated to make it appear like an invisible hand is writing them by setting the stroke style of the letter to be a dashed line, setting the dash length to be the length of the whole stroke, and then changing the dash offset over time. Finally, SVG paths can be used like a parameterized function to find any coordinate along the full path. This is an invaluable feature for the writing feedback methods described in Appendix A that rely on comparing user and "ideal" pen locations.

HAWK activity structure is often repeated across multiple lessons, even if the content changes. For each type of activity HAWK stores two JSON files that contain information about the visual and audio content respectively for that activity and all of its instances. This makes changing activity content a quick process, though changing activity structure is more involved.

## Appendix C - Data Tables

The following tables were too large to insert into the thesis text but are included here for full disclosure.

*Table 0.1: ANOVA analysis results comparing raw time taken to complete lessons between control and experimental groups as part of the KW Class study.*

Lesson #	F	p	Control Average	Control S.D.	Experimental Average	Experimental S.D.
L15	0.56	0.46	117.00	93.50	96.41	110.20
L16	1.01	0.32	278.32	264.44	218.14	180.27
L17	0.95	0.33	400.93	300.64	499.39	442.92
L18	0.03	0.87	434.79	324.44	447.79	251.89
L19	2.53	0.12	244.86	222.43	336.69	213.67
L20	1.37	0.25	354.25	275.23	436.86	257.33
L21	2.74	0.10	178.25	163.49	260.14	206.76
L22	2.19	0.14	447.68	335.58	578.41	331.03
L23	1.81	0.18	459.00	330.19	591.14	394.11
L24	0.36	0.55	462.11	248.27	509.42	320.76
L25	0.40	0.53	146.37	188.57	116.65	152.18

*Table 0.2: ANOVA analysis results comparing normalized time taken to complete lessons between control and experimental groups as part of the KW Class study.*

Lesson #	F	p	Control Average	Control S.D.	Experimental Average	Experimental S.D.
L15	1.37	0.25	74.48	69.50	51.07	78.40
L16	0.18	0.67	184.89	143.90	168.14	150.62
L17	2.43	0.12	235.32	146.11	323.67	260.36
L18	3.16E-07	1.00	364.25	261.46	364.29	211.59
L19	2.88	0.10	171.21	124.80	239.43	172.34
L20	1.20	0.28	238.96	202.35	297.21	195.35
L21	1.24	0.27	105.04	105.81	142.43	142.44
L22	1.35	0.25	299.21	249.32	370.64	208.35

L23	0.74	0.39	327.00	219.75	381.33	242.43
L24	0.26	0.61	345.70	194.11	381.72	309.05
L25	0.22	0.64	121.52	149.91	103.16	133.80

*Table 0.3: ANOVA analysis results comparing student reported affect scores between control and experimental groups as part of the KW Class study.*

Lesson #	F	p	Control Average	Control S.D.	Experimental Average	Experimental S.D.
L15	0.18	0.67	2.96	1.46	2.77	1.45
L16	0.28	0.60	3.38	1.53	3.14	1.55
L17	0.00020	0.99	3.04	1.52	3.05	1.24
L18	0.42	0.52	3.25	1.67	2.95	1.40
L19	0.003	0.96	3.25	1.51	3.23	1.45
L20	0.40	0.53	3.42	1.61	3.14	1.36
L21	2.57	0.12	3.75	1.59	3.05	1.36
L22	0.24	0.63	3.58	1.47	3.36	1.56
L23	3.29	0.08	3.96	1.43	3.19	1.36
L24	5.96	0.02	4.22	1.31	3.16	1.50
L25	0.007	0.94	4.35	1.27	4.32	1.29



## Appendix D - Study Questionnaires

First are the three questionnaires given to the students during their participation in the KW HAWK study. The first and second questionnaire were answered independently by the students. The third questionnaire was given as part of an interview with verbal answers being recorded by the researcher.

Questionnaire about Coding and Kokopelli's World – for February 24, 2016

Name:

Had you every done any coding or computer programming before you started with these lessons?

If so, what was it, and when did you do it? What sort of program did you make?

If so, did you do the coding by yourself, or did you do it with somebody else? Who was it?

What parts of the lessons do you like best? Reading, writing, or coding? Or do you like all parts equally well?

When you are coding, do you prefer small coding problems, or do you prefer to create stories?

If you could have Kokopelli or Raven do anything you wanted to in a story, what would you have them do?

If you could write any kind of story with coding, what kind of story would you tell?

Would you like to share any of your coding stories with your family? Why or why not?

*Figure 0.5: Questionnaire given to students during week eight of the KW HAWK study.*

Questionnaire about Coding and Kokopelli's World – for Week 9, March 2, 2016

Name:

Circle the face that shows how much you like the reading activities.



Circle the face that shows how much you like the writing activities.

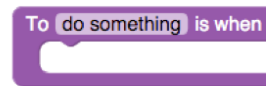
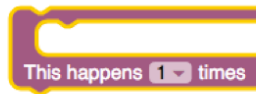
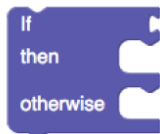


Circle the face that shows how much you like the coding activities.



What did you like about the coding today?

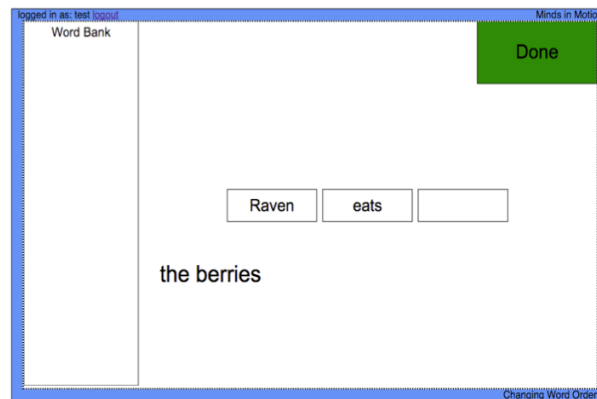
Circle your favorite block to use. Why do you like that block?



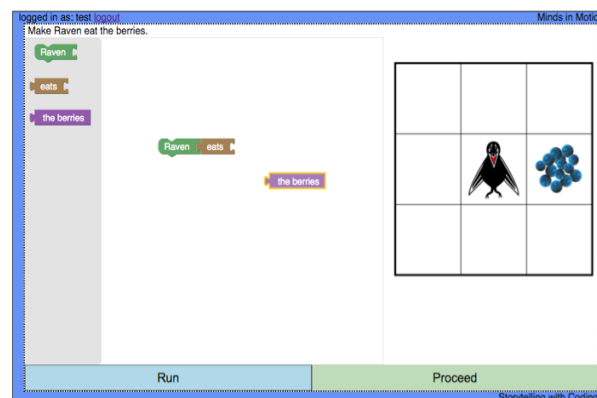

If you wrote your own story for the last coding activity this week, please describe it in writing. If you don't want to write, try drawing a comic about your story.

Figure 0.6: Questionnaire given to students during week nine of the KW HAWK study.

Questionnaire about Coding and Kokopelli's World – for Week 10, March 9, 2016



Do you remember doing this activity?  
 How did you figure this out?  
     What did you do first?  
     Did you do something else after that?  
     How did you know you were done?  
 Why did you do those things?



Do you remember doing this activity? (MIM)  
 How did you figure this out?  
     What did you do first?  
     Did you do something else after that?  
     How did you know you were done?  
 Why did you do those things?

Figure 0.7: Questionnaire given to students in an interview format during week ten of the KW

*HAWK study.*

What follows next is the post-study questionnaire given to students after they had completed the lessons in KW Class. Students filled these questions out individually.

## QUESTIONNAIRE FOR KIDS

Please answer each question, but you don't have to answer a question if you don't want to.

How much fun did you have with the lessons?

☐ Hated It ☐ Didn't Like It ☐ Not Sure ☐ Liked It ☐ Really Liked It

What was your favorite block to use and why?

How useful was the sound in the lessons?

☐ Not at all useful ☐ Not very useful ☐ Not Sure ☐ A little useful ☐ Very useful

How many times did you use the sounds in an average lesson?

If you have programmed before, how did Kokopelli's World compare?

☐ Much worse ☐ A little worse ☐ About the same ☐ A little better ☐ Much better

Would you like to do more lessons in Kokopelli's World?

☐ Yes

☐ No

How did you feel about programming before starting Kokopelli's World?

How do you feel about programming after going through the lessons in Kokopelli's World?

You could say some of the programs you made were like a story, like this one where Kokopelli brings some food for Raven to eat.

**Kokopelli**

**Raven**

grabs

drops

moves south one step




moves west one step

eats

the berries

```

Kokopelli grabs the berries
Kokopelli moves south one step
Kokopelli moves south one step
Kokopelli moves west one step
Kokopelli drops the berries
Raven eats the berries
        
```

While programming in the lessons did it feel like the computer was telling you a story? In what way?

Did it feel like you were writing a story on the computer? In what way?

Did programming feel like you were playing a game? In what way?

Did programming feel like you were making a game? In what way?

**SUBMIT**

Figure 0.8: Screenshots of the KW Class questionnaire filled out by students upon completing the lessons.

Lastly, we share the question script that was given verbally to the students participating in one-on-one sessions attached to the KW Class study. The script is divided into questions that were given before and after experiencing the coding lessons.

### **Child Interview Questions**

- Pre-Treatment:

- § Have you done any programming/coding before?

- Where did you do that?
    - When did you do that?
    - What programming language or system did you use?
    - What did you make with programming?
    - How much did you do? A few minutes? An hour? A bunch of hours?
    - Did you like it?

- § What did you like about it?

- Was there anything you thought was hard about it?
    - Was there anything you didn't like about it?
    - Would you want to do more programming there?

- § In general?

- Do you remember using a block/word having to do with repeating the same thing multiple times?

- § Could you describe in your own words what that block/word does?

- § How did you learn to make that block/word work?

- § Do you remember having any challenges or being surprised by it in any way?

- Do you remember using a block or word having to do with checking if something is true or not?

- § Could you describe in your own words what that block/word does?

- § How did you learn how that block/word works?

- § Do you remember having any challenges or being surprised by it in any way?

- What do you think a programmer does for a job?

- § Something different than what you did before?

- If you could make anything you want with programming, what would you make?

- Post-Treatment:

- § Are there blocks you still aren't sure how they work?

- § Do you remember using the block having to do with repeating something several times?

- Could you describe in your own words what the repeat block does?

- Was that a new idea for you – to tell a computer to do something several times?

- Do you think the software taught you how it works well?

- How did you learn to make it work?

- Do you remember having any challenges or being surprised by it in any way?

- § Do you remember using the block having to do with checking if something is true or not?

- Could you describe in your own words what the if block does?

- Was that a new idea for you – splitting the things a computer *might* do into two parts?

- Do you think the software taught you how it works well?

- How did you learn to make it work?
- Do you remember having any challenges or being surprised by it in any way?
- Did you use the audio to help you on this one?

§ Did you have a lesson or lessons you liked the most?

- Was there any particular lesson/activity that you remember in which you made Kokopelli do something cool/funny/fun?

§ What was it, and why was it cool?

§ What did you not like about the lessons?

- Was there any particular lesson/activity in which you really felt stumped and was not able to figure out how to solve the problem?

§ How did you deal with this?

§ What do you wish you could do with KW/ or wish it had?

§ Did you use any of the block speaker buttons in KW?

- Why/why not?

§ Did it help you?

§ Did you use the “say my code” function?

- Why/why not?

§ Did it help you?

§ Did you notice a difference when the code was read to you in the second half?

- Do you think it helped you in any way?
- Do you think it didn't help you in any way?

§ Would audio features help you if they were part of the programming you did before?

§ Think about reading, writing and math in school, would you say KW programming was more like reading/writing or more like math?

- How so?



§ Do you play video games sometimes?

○ Did it remind you of playing a video game?

§ How so?

§ Do you think a professional programmer would use something like KW?

## Vita

Robert Haven Thompson

### Education

**University of Washington** – GPA: 3.66 September 2006 – March 2011  
Bachelor of Science in Computer Science and Physics with honors.

**University of Washington** – GPA: 3.71 September 2011 – June 2013  
Masters Degree in Computer Science.

**University of Washington** – GPA: 3.75 September 2013 – December 2019  
Doctor of Philosophy in Computer Science.

### Work Experience

**Research Assistant, Paul Allen School of CSE, UW Seattle** June 2010 – December 2019

- Lead developer on multiple web-based projects
- Designed and developed online beginner programming learning environment and set of tutorials
- Designed and developed suite of web-based lessons to teach reading and writing
- Co-Developed online collaborative problem solving and posing environment

**Lecturer/Teaching Assistant, Paul Allen School of CSE, UW Seattle** January 2016 – Dec. 2019

- Sole lecturer for Data Structures course
- Extensive experience teaching AI and database topics (3+ classes for each)
- Taught multiple weekly sections with 20-30 students
- Led multiple coding summer camps with K-12 children
- Experience with Canvas and Gradescope

**Software Development Intern, Zillow, Seattle** June 2011 – September 2011

- Completed two projects presented on main site
- Software generating bird's eye view of property from GPS data
- Work with Maven and Java frameworks

### Programming Expertise

**Web:** JavaScript, PHP, SQL, JSON, SVG, CSS, HTML

**Platforms/Libraries:** CodeIgniter, SVN, Git, D3, Raphael, Drupal

**Other:** Python, Java, C++, (also a little Japanese!)

## Publications

R. Thompson, S. Tanimoto, R. Lyman, K. Geselowitz, K. Begay, K. Nielsen, R. Abbott, W. Nagy, M. Raskind, & V. Berninger. (2017, September 17). *Effective instruction for persisting dyslexia in upper grades: Adding hope stories and computer coding to explicit literacy instruction*. Education and Information Technology.

R. Thompson, S. Tanimoto, R. Abbott, K. Nielsen, K. Geselowitz, R. Lyman, K. Habermann, T. Mickail, M. Raskind, S. Peverly, W. Nagy, & V. Berninger. (2016, July 19, on line). *Relationships between language input and letter output modes in writing notes and summaries for students in grades 4 to 9 with persisting writing disabilities*. Assistive Technology Journal. DOI: 10.1080/10400435.2016.1199066

Thompson, R., Tanimoto, S. *Children's Storytelling and Coding: Literature Review and Future Potential*. PPIG 2016.

Thompson, R., Tanimoto, S., Berninger, V. and William Nagy. *Coding, Reading, and Writing: Integrated Instruction in Written Language*. VL/HCC 2016.

Tanimoto, S., Thompson, R., Berninger, V., Nagy W., Abbott, R. (in press, April 26, 2015). *Computerized writing and reading instruction for students in grades 4 to 9 with specific learning disabilities affecting written language*. Journal of Computer Assisted Learning.

Thompson, R., Tanimoto, S., Berninger, V., William Nagy. *Design Studies for Stylus and Finger-Based Interaction in Writing Instruction on Tablets*. WIPTTE 2015.

Berninger, V., Nagy, W., Tanimoto, S., Thompson, R. Abbott, R., (2015, February). *Computer Instruction in Handwriting, Spelling, and Composing for Students with Specific Learning Disabilities in Grades 4 to 9*. Computers and Education. 81:154–168.

Berninger, V., Tanimoto, S., Thompson, R., Alstad, Z., Griffin, W., Wingert, K., Warmouth, J., (2014, February). *Assessment of ecologically valid writing in middle childhood and adolescence*. In Y.-S. Kim (editor), *Writing assessment from multiple perspectives and for multiple purposes*. Writing Research across the Borders III, Paris, France.

## Presentations

“Teaching Coding to Learning-Disabled Children with Kokopelli's World” – VL/HCC Graduate Consortium 2016

“Children's Storytelling and Coding: Literature Review and Future Potential” – PPIG 2016

“Problem Formulation Affordances for Computer Supported Collaborative Problem Solving” – VL/HCC Graduate Consortium 2015

“Design Studies for Stylus and Finger-Based Interaction in Writing Instruction on Tablets” – WIPTTE 2015