

**Progression of Computational Thinking Skills
Demonstrated by App Inventor Users**

by

Benjamin Xiang-Yu Xie

Submitted to the Department of Electrical Engineering and Computer
Science

in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2016

© Massachusetts Institute of Technology 2016. All rights reserved.

Author
Department of Electrical Engineering and Computer Science
May 23, 2016

Certified by
Harold Abelson
Class of 1922 Professor
Thesis Supervisor

Accepted by
Christopher J. Terman
Chairman, Department Committee on Graduate Theses

Progression of Computational Thinking Skills Demonstrated by App Inventor Users

by

Benjamin Xiang-Yu Xie

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 2016, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

I analyze skill progression in MIT App Inventor, an open, online learning environment with over 4.7 million users and 14.9 million projects/apps created. My objective is to understand how people learn computational thinking concepts while creating mobile applications with App Inventor. In particular, I am interested in the relationship between the development of sophistication in using App Inventor functionality and the development of sophistication in using computational thinking concepts as learners create more apps. I take steps towards this objective by modeling the demonstrated sophistication of a user along two dimensions: breadth and depth of capability. Given a sample of 10,571 random users who have each created at least 20 projects, I analyze the relationship between demonstrating domain-specific skills by using App Inventor functionality and generalizable skills by using computational thinking concepts. I cluster similar users and compare differences in using computational concepts.

My findings indicate a common pattern of expanding breadth of capability by using new skills over the first 10 projects, then developing depth of capability by using previously introduced skills to build more sophisticated apps. From analyzing the clustered users, I order computational concepts by perceived complexity. This concept complexity measure is relative to how users interact with components. I also identify differences in learning computational concepts using App Inventor when compared to learning with a text-based programming language such as Java. In particular, statements (produce action) and expressions (produce value) are separate blocks because they have different connections with other blocks in App Inventor's visual programming language. This may result in different perceptions of computational concepts when compared to perceptions from using a text-based programming language, as statements are used more frequently in App Inventor than expressions.

This work has implications to enable future computer science curriculum to better leverage App Inventor's blocks-based programming language and events-based model to offer more personalized guidance and learning resources to those who learn App Inventor without an instructor.

Thesis Supervisor: Harold Abelson
Title: Class of 1922 Professor

Acknowledgments

I thank members of MIT Center for Mobile Learning (encompassing MIT App Inventor, Scratch, and STEP Labs) for their knowledgeable feedback and unwavering support in my work. Specifically, I thank my adviser Hal Abelson for holding me accountable, Sayamindu Dasgupta for providing fresh insight whenever I was blocked, Aubrey Colter for being my go-to editor, and Nicole Zeinstra for assuring me the sky was never actually falling.

I hope that this thesis as a humble grain of contribution to the ever-growing beach that is Computing Education Research.

Contents

1	Introduction: Measuring demonstrated skills in an open environment	15
1.1	The question: How do people learn CS skills by creating apps?	16
1.2	MIT App Inventor democratizes the creation of mobile apps	17
1.2.1	App Inventor programs respond to events	17
1.2.2	App Inventor is used outside of classrooms	19
1.3	Computational concepts exist in many programming languages	20
1.4	Using App Inventor functionality and using computational concepts .	21
1.5	Thesis Overview	21
2	Method: Modeling the progression of demonstrated skills	25
2.1	Data consists of long-term users' projects	25
2.1.1	Measuring block types of active blocks prevents double counting	27
2.2	Usage of new block types models breadth of capability	29
2.3	Block frequency, events handled measures depth of capability	31
2.4	Comparing skills using App Inventor functionality and skills using computational concepts	32
2.4.1	Computational concept blocks are independent of components	33
2.4.2	Events are counted separately	33
2.5	Clustering is based on IDF-weighted computational concepts	33
3	Results	37
3.1	Frequency of CC Blocks	37

3.2	Breadth of capability begins to plateau	38
3.3	Depth of capability continues to increase	42
3.4	Clustering to identify CC usage patterns	42
3.4.1	Representative Users	47
4	Discussion	51
4.1	Developing breadth before developing depth of skill	52
4.2	Comparing to Scratch	53
4.3	Learning programming with blocks in App Inventor \neq learning with text languages	54
4.3.1	The connections in block languages separate statements from expressions	55
4.3.2	Computational concepts may develop from manipulating com- ponents	57
4.3.3	Measuring complexity of computational concepts relative to App Inventor's event-based model	58
4.4	Limitations	60
4.4.1	Measuring blocks is not enough	60
4.4.2	End-user programmers care less about computational thinking	60
4.5	Future Work: Data on project progression, users would extend work .	61
5	Conclusions	63
5.1	Implications	63
5.1.1	Teachers can develop curriculum with App Inventor's event- based environment in mind	63
5.1.2	Researchers can quantitatively measure progression of skill in blocks-based environments	64
5.1.3	App Inventor learners can measure their progression of learning	64
5.1.4	Contributions	65

A Related Work: Computational Thinking Frameworks, Measuring

Demonstrated Skill	67
A.1 Breadth and depth are measures of demonstrated skill	68
A.1.1 Learning trajectories model the <i>breadth</i> of capability	68
A.1.2 The number of block types in a project measure the <i>depth</i> of capability	70
A.2 Computational Concepts are a dimension of Computational Thinking	70
A.3 Blocks languages are not perceived the same as text languages	71
B Description of Computational Concept (CC) Block Types	75

List of Figures

1-1	Estimated time spent in school and informal learning environments [24]	16
1-2	Example script for Blockly programming language in MIT App Inventor.	18
1-3	Step from written tutorial for creating Magic-8 ball app [26].	20
2-1	Example of how block types are counted. Block types, not block frequencies are counted for blocks connected to top blocks.	28
2-2	Built-in blocks are component-independent. Here, some of the built-in blocks from the Control category are showing (<code>controls_if</code> , <code>controls_forEach</code>).	29
2-3	Blocks from the 6 computational concepts in App Inventor. Clockwise from top left: procedure, variable, logic, loop, conditional, list.	32
3-1	Histogram of Computational Concept (CC) block types	39
3-2	Cumulative number of new blocks introduced at a given project . . .	40
3-3	Normalized cumulative number of new blocks introduced at a given project	41
3-4	Normalized average rate of introducing new block types to projects .	41
3-5	Average number of block types used in each project (to measure depth of capability)	43
3-6	Average number of events responded to in each project (to measure depth of capability)	43
3-7	Selecting number of clusters by considering average within-cluster sum squared error. $k = 2$ is selected.	44
3-8	Average cumulative sum of IDF-weighted CC block types for each cluster	45

3-9	Normalized cumulative sum of weighted CC block types for each cluster of users	46
4-1	Procedures without return values (top) and with return values (bottom).	56
4-2	If/else blocks to determine which statements to execute (left) and which expressions to return (right)	56
4-3	The <code>lexical_variable_get</code> block (in orange) can access both com- ponent parameters (<code>x</code> , <code>y</code>) as well as global variables (<code>dotsize</code>)	57
A-1	Blocks from Scratch, an environment similar to MIT App Inventor . .	67
A-2	Results from Scaffidi [19] showing a decrease in average breadth and depth in Scratch projects over time.	69
A-3	Student reported differences between Snap! and Java at mid-point and conclusion of study (from [27]).	72

List of Tables

3.1	Information on Clusters	45
3.2	Representative Users' Orders of Acquiring Computational Concepts, Cluster 0	49
3.3	Representative Users' Orders of Acquiring Computational Concepts, Cluster 1	50
B.1	Description of Variable Blocks	75
B.2	Description of Procedure Blocks	76
B.3	Description of Loop Blocks	76
B.4	Description of Logic Blocks	77
B.5	Description of Conditional Blocks	77
B.6	Description of List Blocks	78

Chapter 1

Introduction: Measuring demonstrated skills in an open environment

The objective of this thesis is to understand how people learn computational thinking concepts, concepts which exist in many programming domains (such as procedures and conditionals), while creating apps with App Inventor's open programming environment. I model the sophistication of demonstrated skill using two dimensions: Breadth of capability and depth of capability. My data consists of project data from 10,571 App Inventor users who have each created at least 20 projects (a total of 211,420 projects). With this, I analyze the relationship between learning domain-specific skills (learning App Inventor functionality) and learning skills that generalize to other programming domains (computational concepts). I also cluster similar users and identify a common pattern in acquiring new computational concept skills that is influenced by App Inventor's block-based programming language and events-based model.

This is the issue being addressed: That we must understand how people learn computational thinking skills with App Inventor so that these open, informal experiences of creating mobile applications with App Inventor can be integrated with STEM curricula. By quantitatively modeling the use of computational thinking concepts,

we are able to measure and monitor the development of computational thinking skills at scale. This information can help develop computer science curriculum to leverage App Inventor’s unique environment and also provide a more personalized experience for those who learn programming with App Inventor on their own.

1.1 The question: How do people learn CS skills by creating apps?

At any given age in our life, we spend less than 20% of our time in a formal learning environment, as shown in Figure 1-1. I am interested in understanding how people learn when they are not in classrooms or lecture halls, when they are in the *sea of blue* that is informal learning environments. It is my intention to understand these self-guided informal learning experiences to improve them and improve curriculum taught in formal learning environments.

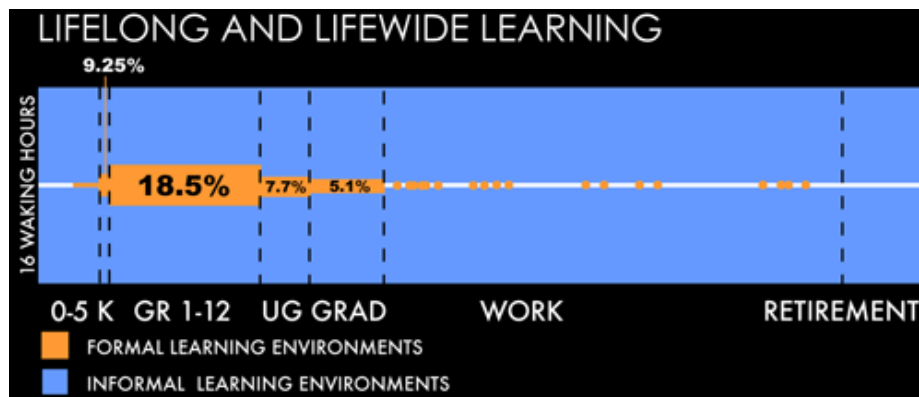


Figure 1-1: Estimated time spent in school and informal learning environments [24]

The onset of the internet and ubiquitous computing has enabled people to learn programming outside of formal learning environments through accessible online resources. 4.7 million people have tried programming with MIT App Inventor, an online environment that leverages a blocks-based visual language (Blockly, [3]) to enable people to create mobile applications (apps) [15].

This thesis asks the following questions:

- How do people develop skills using computational concepts when creating apps with App Inventor?
- What is the relationship between developing domain-specific skills of using App Inventor functionality and developing generalizable skills using computational concepts?
- What is the typical order for acquiring computational concepts in App Inventor?

1.2 MIT App Inventor democratizes the creation of mobile apps

A project/app made with App Inventor consists of a set of components and a set of program blocks that provide functionality to these components. Components include items visible on the phone screen (e.g. buttons, images, text boxes) as well as non-visible items (e.g. camera, database, sensors). Figure 1-2 shows blocks used in an app that automatically responds to text messages and reads them aloud.

App Inventor has reached a broad international audience for use both in and out of classrooms and formal learning environments. As of May 2016, 4.7 million people from 195 countries have created over 14.9 million apps [15]. App Inventor is taught to a broad audience, ranging from grade school to college students. Industry professionals also use App Inventor, often as end-user developers who write programs to support their primary work or hobbies [10]. I discuss differences in objectives for end-user programmers when compared to typical people who want to learn programming in section 4.4.2.

1.2.1 App Inventor programs respond to events

App Inventor is an events-based model where even introductory projects involve specifying how the app should respond to events related to device features [25]. Examples include responding to receiving a text message, pressing a button, touching the screen,

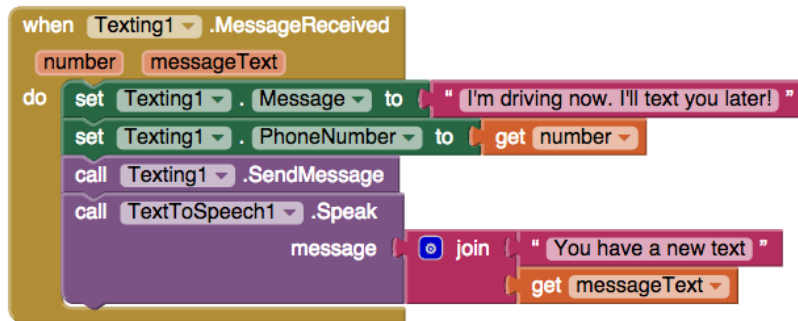


Figure 1-2: Example script for Blockly programming language in MIT App Inventor.

or shaking the device. This events-based model is made possible by providing event-handling blocks (e.g. when text received in Figure 1-2) for all events that exist for any component. Components have specific blocks that respond to events, set/get properties, and call methods.

In App Inventor, all functionality is initiated by event handling blocks, referred to later as *top blocks*. These event-handling blocks are associated with components. In Figure 1-2, the `Texting.MessageReceived` block relates to the Texting component. Most blocks in App Inventor are component-specific. These event handling blocks may also have parameters; In Figure 1-2, the `Texting.MessageReceived` event handler block has parameters that pass in the phone number of the message sender and the contents of the text message.

In addition to having event-handling blocks, App Inventor components also have blocks to access and define properties. Component properties are attributes to the component, such as the text in a label or the width of an image. A texting component's properties include the message to send and the phone number to send the message to. Blocks exist to get and set these properties (green blocks in Figure 1-2 set properties).

Blocks also exist to call component-specific methods. Whereas more basic components only have event-handlers and properties (e.g. buttons, labels), others have additional functionality. Examples include sending a text message with a Texting component or reading text aloud with a TextToSpeech component, as shown in Figure 1-2.

The events-based model makes learning to program with App Inventor unique as some concepts are easier to learn than others with this model. In example, users learn to access component properties and event parameters early on and will tend to use conditional statements (if/then) in earlier projects. In contrast, learners tend to use loops less frequently because few events require iteration. I discuss this further in section 4.3.

1.2.2 App Inventor is used outside of classrooms

App Inventor is often used outside of formal learning environments. An online survey of 221,771 self-selected users between October 2013 and April 2016 revealed that 70% of people use App Inventor in their homes. Decreases in usage during school holidays also support the fact that a significant portion of people use App Inventor outside of formal learning environments.

This self-paced and self-motivated pathway of learning to program is distinct and different from the traditional classroom settings. Interactions with peers often do not occur in person for these self-learners. Instructors often do not exist, resulting in a need for online learning resources. Environments vary and the pace of learning is not as rhythmic as a semester or quarter.

The primary learning resources for these self-directed App Inventor learners are step-by-step tutorials. These tutorials guide users through creating an entire functioning app from start to finish. Each tutorial typically focuses on either introducing a new component (such as a canvas or GPS integration) or additional functionality for a previously introduced component. The App Inventor resources page on the App Inventor website contains 26 tutorials ranging from beginner level to advanced difficulty [15]. App Inventor’s open source community has created copious other learning resources and curricula that have become popular as well. Examples include App Inventor’s Course In a Box [1], Mobile Computer Science Principles [16], and Computing Science Principles for High School Teachers [17].

3. From the Built-In palette, click on the Lists drawer. Drag over the `pick random item` block and connect it to the open socket of the `set Label2.Text` block.

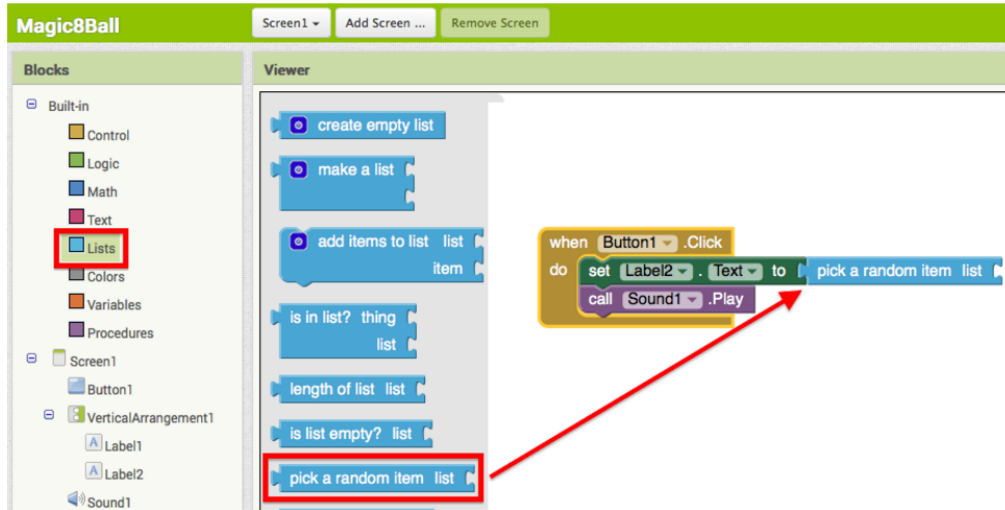


Figure 1-3: Step from written tutorial for creating Magic-8 ball app [26].

1.3 Computational concepts exist in many programming languages

Computational thinking was first defined in 2006 by Jeannette M. Wing: "Computational thinking involves solving problems, designing systems, and understanding human behavior, by drawing on the concepts fundamental to computer science" [28]. Since then, it has been the focus of much computing education research as computational thinking reflects skills that generalize across domains in computer science as well as other fields. At the core, computational thinking is about abstraction. Quoting Wing: "Abstraction is used in defining patterns, generalizing from instances, and parameterization. It is used to let one object stand for many. It is used to capture essential properties common to a set of objects while hiding irrelevant distinctions among them" [29].

A framework for studying and assessing computational thinking was developed by Brennan 2011 ([5]) for Scratch, a visual blocks-based programming environment used to create media projects [21]. This framework describes computational concepts, the concepts people engage with as they program, as a key dimension of computational thinking. These concepts are widely used and common in many programming lan-

guages. (More information on computational thinking and the CT framework can be found in section A.2)

I analyze the use of computational concepts across projects to understand how learners develop their skills to use these concepts that can generalize to other programming domains.

1.4 Using App Inventor functionality and using computational concepts

Learning to use App Inventor is not useful unless the skills users learn are relevant beyond the App Inventor environment. App Inventor is intended to be an introductory programming experience that serves as a springboard to future programming in other contexts and domains. For this reason, I am curious how the development of domain-specific skills of using App Inventor relates to the development of generalizable skills of using computational concepts. If users develop their capabilities to use App Inventor functionality in a similar fashion to how they develop their capabilities to use computational concepts, we can say that App Inventor users learn computational concepts as they learn to use App Inventor. This would suggest that App Inventor is a beneficial introductory experience because it prepares new programmers for future coding experiences with other languages in other environments.

1.5 Thesis Overview

In this thesis, I conduct two experiments: 1) I measure the breadth and depth of capability as evidenced by projects that users have created to understand the development of skills pertaining to computational concepts. 2) I cluster users who utilize computational concepts similarly and qualitatively analyze users close to the centroid of each cluster to determine the order learners acquire computational concepts.

In chapter 2, I detail my methodology for my experiments. I elaborate on my data source of projects from 10,571 long-term users and explain how I measure block types

and why measuring block types is more effective than simply measuring the number of blocks. I then explain how I calculate the breadth of capability by considering the number of new block types introduced at each project and how to calculate the depth of capability by considering the total number of block types used at each project. I then elaborate on how I separate the block types into two disjoint sets based on whether the block relates to a computational concept or not. I close with explaining how I use Inverse Document Frequency (IDF) weighting to cluster learners who used computational concepts in similar fashions.

In chapter 3, I describe my results and findings. I find that computational concept blocks pertaining to list operations and loops/iterators are not commonly used in App Inventor, perhaps because of App Inventor's event-based model for programming. The breadth of learning decreases as users create more projects, as the depth of learning continues to increase. Looking at representative users reveals similarities in the order learners first use computational concepts.

In chapter 4, I discuss the results in the context of other blocks-based environments and text-based environments. I propose a behavior of "breadth before depth" where users tend to familiarize themselves with a wide array of components and block types in earlier projects and then develop a mastery of previously learned skills in later projects. I hypothesize that App Inventor users' depth of capability continually increases over time because App Inventor is a robust and extensible environment so it is still engaging to advanced and long-term users. I note that connections between programming blocks differentiate statements (code that produces an action) from expressions (code that produces a result). This makes learning to program with App Inventor unique when compared to learning to program with a text-based programming language such as Java. I propose a 3-phased progression of complexity for computational concepts that is based on how users interact with components (access component properties, use component methods, change component state). I also note limitations and future work.

In chapter 5, I conclude by noting implications for different interested parties and stating the contributions of my work. Teachers can use these findings to develop a

computer science curriculum that leverages App Inventor’s unique blocks-based programming language and events-based model. Researchers can use this methodology to quantitatively model the development of generalizable computational concept skills and the results to draw comparisons between environments. Students can track their learning progress with the computational concept complexity measure proposed in section 4.3.3 and future tools may use a similar concept complexity measure to recommend tutorials and learning resources for self-learners based on the computational concepts they used (or did not use) in previous projects.

Appendix A contains related work pertaining to computational thinking, measuring sophistication, and perceptions between blocks and text languages. Appendix B explains the computational concept block types.

Chapter 2

Method: Modeling the progression of demonstrated skills

In this chapter, I explain my technical approach. I elaborate on my data source, project data from long-term App Inventor users (section 2.1). I model sophistication of projects using two dimensions: breadth of demonstrated capability (section 2.2) and depth of demonstrated capability (section 2.3). These dimensions of measuring sophistication of end-user computing were first defined by Huff 1992 [9] and I elaborate on this in section A. I separate computational concept (CC) blocks from non-CC blocks to draw comparisons between demonstrating domain-specific skills by using App Inventor functionality and demonstrating generalizable skills by using computational concepts (section ch2:comparing). Finally, I explain how I cluster similar users and use cluster centroids to understand typical patterns of using computational concepts (section 2.5).

2.1 Data consists of long-term users' projects

My data source consists of project data from 10,571 random App Inventor users who have created at least 20 projects. Our data sources show that the top 1.4% of App Inventor users (approx. 65,000) have created at least 20 projects. I only look at the first 20 projects of a user, ignoring any further projects created. In total, I analyze

211,420 projects created by 10,571 users randomly selected from the subset of App Inventor users who have created at least 20 projects. The projects were created between 27 March 2013 and 10 March 2016. Each project consists of the following files:

- *META*: Contains time project was created and last modified
- **.bky*: Data on blocks for a given screen (as XML)
- **.scm*: Data on components for a given screen (as JSON)
- *project.properties*: Contains project name (as well as other information extraneous to this thesis)

I worked with Professor Franklyn Turbak (Department of Computer Science, Wellesley College) and his student Maja Svanberg to write a python script to summarize the project data into a JSON format. We extract the following data from the project files:

- Project name
- Time of creation
- Time of last modification
- Media asset file names
- Data for each screen in a project:
 - Blocks info:
 - * Top level blocks: type and frequency of event handler, variable definition, and procedure definition blocks
 - * Active blocks: type and frequency of all blocks within top level blocks (blocks that have functionality), names of procedures and variables and number of times each was called
 - * Orphan blocks: type and frequency of all blocks not connected to a top level block (no functionality)
 - Component information: Type and frequency of components in screen

I use this data in each project summary to represent the project in my analysis. Of particular note is that the project summary primarily contains information pertaining to what was created and almost no information that identifies the user. Because a

significant portion of App Inventor’s user base is under 18 (27% of users according to a self-selected survey with 238,065 responses), App Inventor does not collect user information. App Inventor users are only identified via their Google account for signing in. So, this thesis focuses on what people create while knowing almost nothing about the people. Considering user information would be opportunities for future research as described in 4.5.

2.1.1 Measuring block types of active blocks prevents double counting

For my analysis, I consider all block types for blocks that may have functionality. There are three things I wish to make clear about measuring block types: 1) Counting block types are not the same as counting blocks; 2) I only consider blocks that are connected to top blocks and therefore may have functionality; 3) There are many block types because most relate to component-specific functionality.

I count block types rather than blocks to prevent double counting. I present a basic example to explain this: Suppose Alice and Eve create two apps of identical functionality. Alice uses a procedure and calls the procedures so three buttons in her app have similar functionality. In contrast, Eve does not use a procedure and merely copies and pastes code between her three buttons. I argue that Alice has demonstrated greater skill because she has used a procedure to prevent redundant code and make her project more ready for future change. Therefore, Alice’s project should be seen as more sophisticated. Eve’s project has more blocks because she copies and pastes code, but Alice’s project has more block types because she uses procedure definition and call blocks. So, counting the number of block types prevents double counting which can result from poor programming habits and is therefore a more ideal metric of analysis. By counting block types, I ignore the number of different procedures or variables defined in a given project.

A potential limitation to counting block types is that I do not consider the number of procedures or variables defined within a project. This is because all procedures

are defined by `procedure_defnoreturn` and `procedure_defreturn` blocks (depending on if they return a value or not) and all global variables are defined by the `global_declaration` blocks. This is a potential limitation to counting block types.

Omitting blocks that are not connected to top blocks ignores blocks that certainly have no functionality, a source of noise. Previous work in quantitatively analyzing App Inventor projects noted that a source of noise in the data was that blocks not connected to top or header blocks were being counted (Xie 2015 [30]). Working with Professor Turbak and Maja Svanberg, I was able to separate blocks that were not attached to top blocks and omit them from my analysis.

Figure 2-1 shows an example of how block types for active blocks are counted. Here, we count 3 block types: `Button.Click` (the top block that handles the button click event), `procedures_callnoreturn` (a call to procedure named "my_procedure"), and `Player.Start` (a block to play audio from a player component named "MusicPlayer"). The second procedure call is not counted again because we are considering unique block types. The `lists_is_in` block (to determine if a list contains a specific entry) is not connected to a top block so it has no functionality and is therefore also not counted.

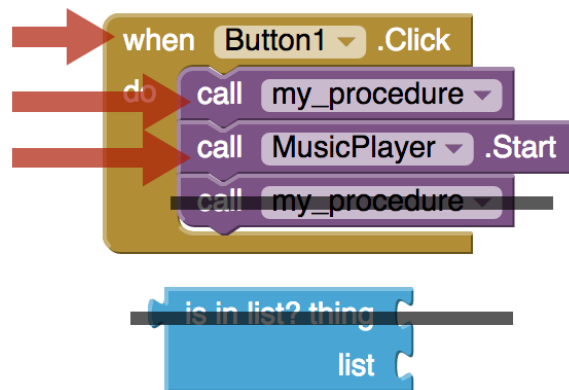


Figure 2-1: Example of how block types are counted. Block types, not block frequencies are counted for blocks connected to top blocks.

There are 1,333 distinct block types in my dataset. 1,241 block types (93%) are component specific. The other 92 block types (7%) are not specific to any components and are known as *built-in* blocks. They are separated in 8 categories: Control, Logic,

Math, Text, Lists, Colors, Variables, Procedures. Figure 2-2 shows the categories of the built-in blocks in the workspace. (See 1.2.1 for more on component-specific blocks.)

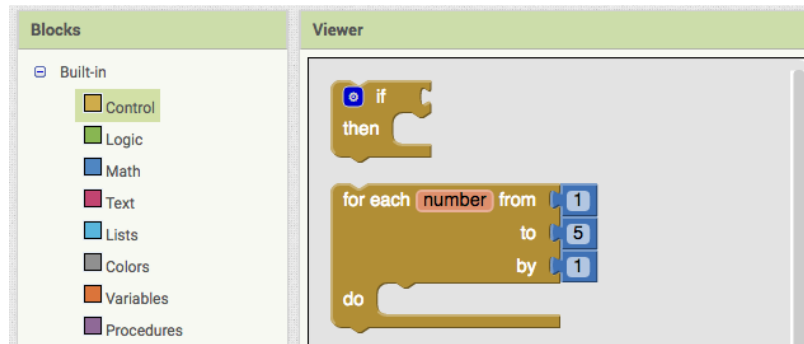


Figure 2-2: Built-in blocks are component-independent. Here, some of the built-in blocks from the Control category are showing (`controls_if`, `controls_forEach`).

2.2 Usage of new block types models breadth of capability

The first dimension of sophistication I consider is the *breadth* of capability as evidenced by what users create. Breadth of capability reflects the broad understanding of knowledge and skill that users demonstrate. **I model breadth of capability as the number of new block types used in each of a user’s projects.**

I adapt the concept of a learning trajectory as originally defined for Scratch by Yang 2015 [31] to measure cumulative breadth of capability for a user across their first 20 projects. (See section A.1.1 for related work on learning trajectories.)

To model the breadth of capability, I do the following:

1. For Each User:
 - (a) Isolate a specific set of block types, S . For my analysis, I choose the sets to be computational concept (CC) blocks and non-CC blocks. These sets are disjoint (CC blocks explained in section 2.4.1).
 - (b) Create matrix P_{user} , which is the frequency of each type of block in each project. Each row is a project a user has created (in sequential order by

- creation time) and each column is the frequency of a certain block type.
- (c) Use P_{user} to create P_{cum} , the cumulative sum of P_{user} .
 - (d) Use P_{cum} to create P_{binary} which is an indicator matrix (1 if certain block has been used by project i, 0 otherwise).
 - (e) Create the trajectory $V_{breadth}$ by summing the values in each row of P_{binary} (summing the new block types used for the first time in a given project).
2. Calculate T_{CC} (or T_{non-CC} depending on S) where each row is $V_{breadth}$ for a particular user. Each row of this matrix reflects the cumulative number of new block types introduced up to a given project for a user.
 3. Calculate the difference matrices $T_{diff,CC}$ (or $T_{diff,non-CC}$) by finding the first order difference of values between columns. These difference matrices measure the acquisition rate, or number of new block types used for the first time at each project.

A notable difference in the adaptation of learning trajectories for use in App Inventor is that I consider all blocks of equal weight when defining trajectories. Yang 2015 uses Inverse Document Frequency (IDF) block weighting (IDF: [23]) to assign greater weight to blocks that reflect greater sophistication [31]. In example, a Scratch block to set a value in a list (`setline_oflist_to`) is weighted higher than an if conditional block (`doif`). This was found to be effective for Scratch, which has a relatively small total corpus of 170 block types. In comparison, the data I analyzed in App Inventor yield a total corpus of 1,333 different block types. As stated in section 2.1.1, most of these block types pertain to events of different components. Because App Inventor’s extensive features set, IDF weighting would assign greater weight to blocks relating to rarely used functionality, rather than assign greater weight to blocks requiring more sophistication to use (as intended). I do consider IDF weighting when clustering similar users (see section 2.5), but not when measuring breadth or depth of capability.

2.3 Block frequency, events handled measures depth of capability

The second dimension of sophistication I consider is the *depth* of capability as evidenced by what users create. Depth of capability refers to the mastery of certain features and functions. **I model depth of capability as the *total* number of block types used in each of a user's projects.**

To model the depth of capability, I do the following:

1. For Each User:
 - (a) Isolate a specific set of block types, S . For my analysis, I choose the sets to be computational concept (CC) blocks and non-CC blocks. These sets are disjoint. (explained in section 2.4.1)
 - (b) Create matrix P_{exist} , which checks for the existence of each type of block in each project (1 if in project, 0 otherwise). Each row is a project a user has created (in sequential order by creation time) and each column is the frequency of a certain block type.
 - (c) Create the trajectory V_{depth} by summing the values in each row of P_{exist} (summing the total number of block types used in each project).
2. Calculate D_{CC} (or D_{non-CC} depending on S) where each row is V_{depth} for a particular user. Each row of this matrix reflects the number of block types used in a given project for a user.

I also consider the number of events responded to in a project as another metric of measuring depth of capability. The number of events handled is part of the project summary, so I extract the number of top blocks in a project (excluding variable and procedure definitions) to measure the number of events responded to.

2.4 Comparing skills using App Inventor functionality and skills using computational concepts

Computational concept (CC) blocks and Non-CC blocks are disjoint sets of block types. I use the set of CC blocks to understand how learners use generalizable computational concepts and use the set of non-CC blocks to understand how learners demonstrate use of App Inventor functionality. The objective is to compare the progression of computational concept skills with the progression domain-specific App Inventor skills.

Computational concepts are part of a computational thinking framework defined for Scratch by Brennan 2011 [5] (see section A.2 for more on computational thinking and the CT framework). I adapt it for use in App Inventor by defining 6 computational concepts: procedure, variable, logic, loop, conditional, and list. Blocks from each category are shown in Figure 2-3. In total, there are 39 CC block types. I separate them from the other block types to create the disjoint sets of CC blocks and non-CC blocks. The set of non-CC block types has 1,294 block types, most of which are component-specific (as explained in 2.1.1). Appendix B provides further information on the 39 CC block types.

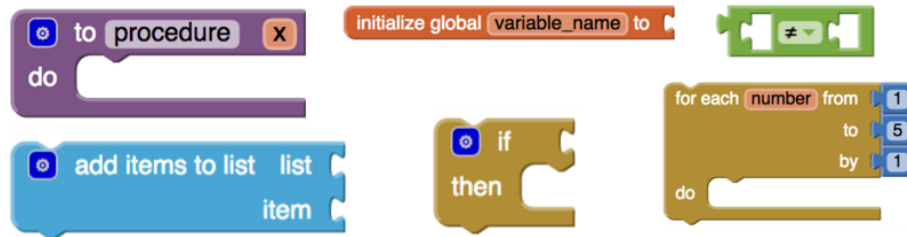


Figure 2-3: Blocks from the 6 computational concepts in App Inventor. Clockwise from top left: procedure, variable, logic, loop, conditional, list.

2.4.1 Computational concept blocks are independent of components

I only consider built-in blocks when determining which blocks are computational concept (CC) blocks because computational concepts are generalizable across programming domains. That is, I do not consider any component-related blocks when identifying which blocks are CC blocks. This is to ensure that a project’s functionality (as determined by the components used in that project) do not enable or limit the use of CC blocks. My findings suggest that CC blocks do not reflect all generalizable skills that learners use in App Inventor, a limitation detailed in section 4.4.

2.4.2 Events are counted separately

I consider events separately from other computational concepts because they are so ubiquitous and essential to programming with App Inventor. Events are another computational concept mentioned in Brennan’s framework for assessing computational thinking in Scratch. As mentioned in section 1.2.1, creating apps with App Inventor involves programming responses to component events. So, any App Inventor project with functionality will have responded to at least one event. Because event-handler blocks are different block types for each event in each component, event-related blocks would dominate CC block counts such that blocks related to other computational concepts would be almost irrelevant in the measurement. I consider the number of events responded to separately as a measurement of the depth of capability users demonstrate (as defined in section 2.3).

2.5 Clustering is based on IDF-weighted computational concepts

I cluster similar users and look at those nearest to the center of each cluster to analyze common patterns of using computational concept skills and compare differences between them. I cluster based on the sum of the number CC block types used at a given

project, where each CC block type is inverse document frequency (IDF) weighted. I use the K-Means clustering algorithm to cluster users [13], determining an optimal number of clusters by looking at the average within-cluster sum of squared error for each number of clusters (k) in a reasonable range (referred to as the elbow method [11]). Finally, I look at the 3 users nearest to each cluster center and determine similarities in CC use between users in the same clusters and differences across clusters.

In the context of this thesis, inverse document frequency measures how much information a CC block provides towards the sophistication of user capability [23]. That is, IDF weighting determines whether a block is common or rare across projects. The IDF weighting for a CC block b is obtained by dividing the total number of projects N by the number of projects p from the total corpus of sampled projects C that contain the block b and then taking the logarithm of that quotient. The equation for IDF weighting:

$$idf(b, C) = \log \frac{N}{|\{p \in C : b \in p\}|}$$

To calculate the feature vector to cluster on, I follow the same process as calculating the matrix for modeling depth of capability (see section 2.3) except I apply the IDF weight vector W to P_{exist} before values in each row are summed. The result is a matrix where each row represents a user and each column represents a given project. The value at the i -th row and j -th column is the sum of the IDF weighted CC blocks used in user i 's j -th project.

With this feature vector, I run K-Means to cluster users who use CC blocks similarly. I use the Scikit Learn implementation of K-means clustering (sklearn: [20]). Initialization is done with K-means++ with 10 different random initial centroid seedings. I use the *elbow method* to determine the number of clusters by analyzing the average within-cluster sum squared distance between points in the cluster and the centroid. After selecting my optimal number of clusters, I run K-means again and then select the three users closest to the cluster center for each cluster. I say these

users represent the cluster and find similarities between representative users within the same cluster and identify differences between clusters.

Chapter 3

Results

I describe my results and findings. I find that computational concept blocks pertaining to list operations and loops/iterators are not commonly used in App Inventor, perhaps because of App Inventor's event-based model for programming. The breadth of learning decreases as users create more projects, as the depth of learning continues to increase. Looking at representative users reveals similarities in the order learners first use computational concepts.

3.1 Frequency of CC Blocks

Figure 3-1 shows the number of projects (from the sample of 211,420 projects) that contain each of the 39 computational concept blocks. Notable findings:

- The five most common CC blocks get a variable (`lexical_variable_get`), define a variable (`global_declaration`), provide an if statement conditional (`controls_if`), set a variable (`lexical_variable_set`), and provide a boolean (`logic_boolean`).
- The least used half of the blocks relate to list operations, if/else conditional expressions (`controls_choose`), or operators (`logic_or`), while loops (`controls_while`), and local variable declarations (`local_declaration_expression`).
- More procedures are defined than called. This is likely because a procedure

definition block is a top block and counted in our analysis even if the procedure is empty and does nothing. In contrast, a procedure call block would have to be connected within a top block to be counted. (See section 2.1.1 for more on top blocks.)

- Only 15% of procedures return values. This is likely because the App Inventor environment lends itself to using procedures to share functionality across components (e.g. having 3 buttons changing the color of a brush in a painting app). So, a procedure without a return value may be more useful in the App Inventor environment. I discuss this in more detail in section 4.3.1.

3.2 Breadth of capability begins to plateau

I show how the progression of the breadth of capability to use App Inventor functionality relates to the breadth of capability to use computational concepts by measuring the number of new blocks introduced at each of a user's projects. Figure 3-2 shows the cumulative sum of block types introduced at each project, averaged over all 10,571 users. CC blocks and non-CC blocks are shown as separate trajectories.

Given that there are a total of 39 CC blocks and 1,294 non-CC blocks, the divergence of CC and non-CC trajectories is expected. We see that the rate of new block acquisition decreases as users create more projects, showing that users introduce new types of blocks at a decreasing rate as they create more projects. It is also of note that even by the 20th project (the last project measured in this analysis), the average number of CC and non-CC blocks used is nowhere near the total number of CC and non-CC blocks. This suggests that users are not exploring all of App Inventor's functionality or computational concepts and therefore are *not* being upper-bound by the limitations of the App Inventor environment.

I normalize the trajectories of block type acquisition in Figure 3-3. In this figure, a diagonal trajectory would suggest users introduce new block types at a constant rate across all projects. From this, we see that the progression of developing skills to use App Inventor functionality and the progression of developing skills to use com-

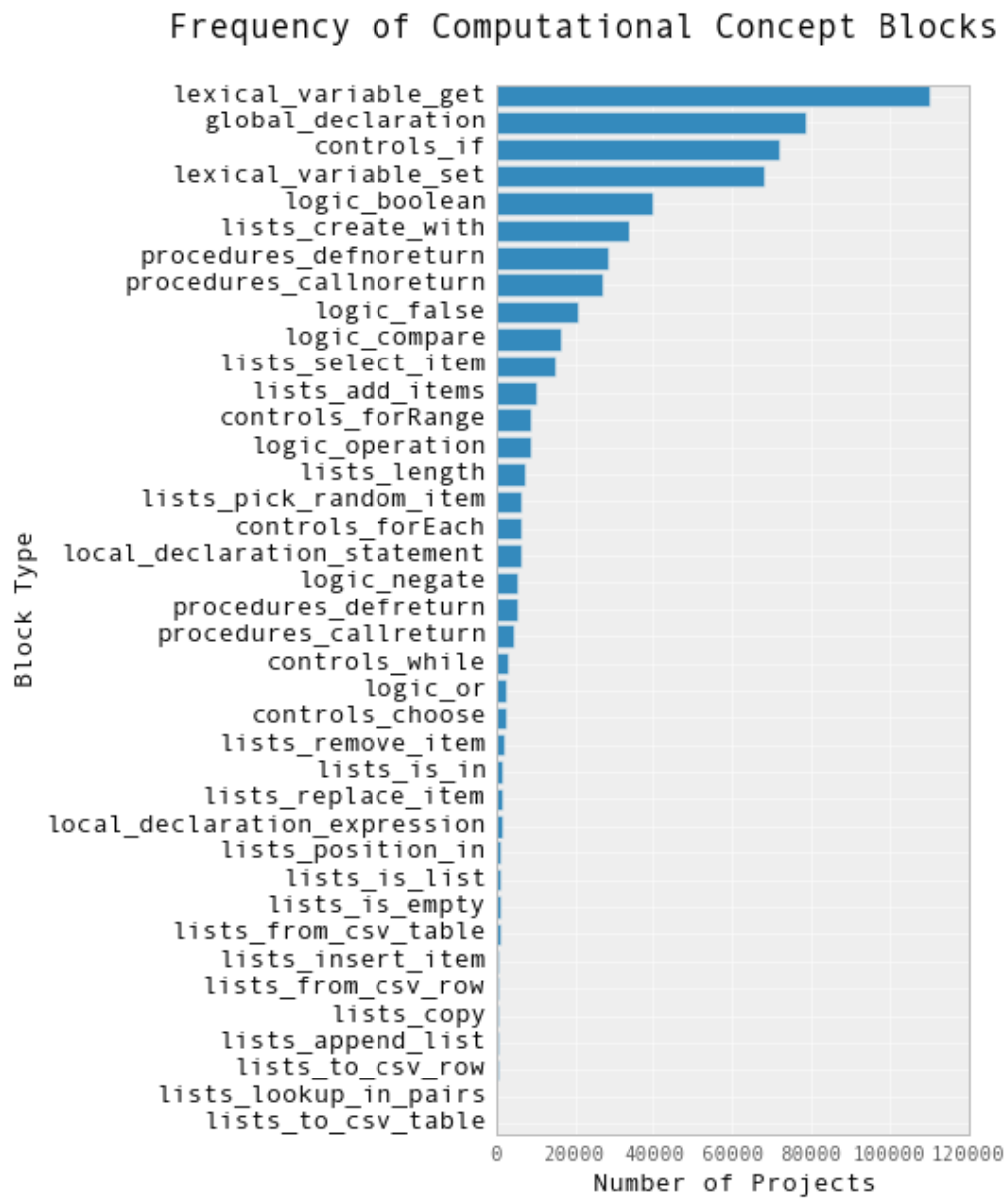


Figure 3-1: Histogram of Computational Concept (CC) block types

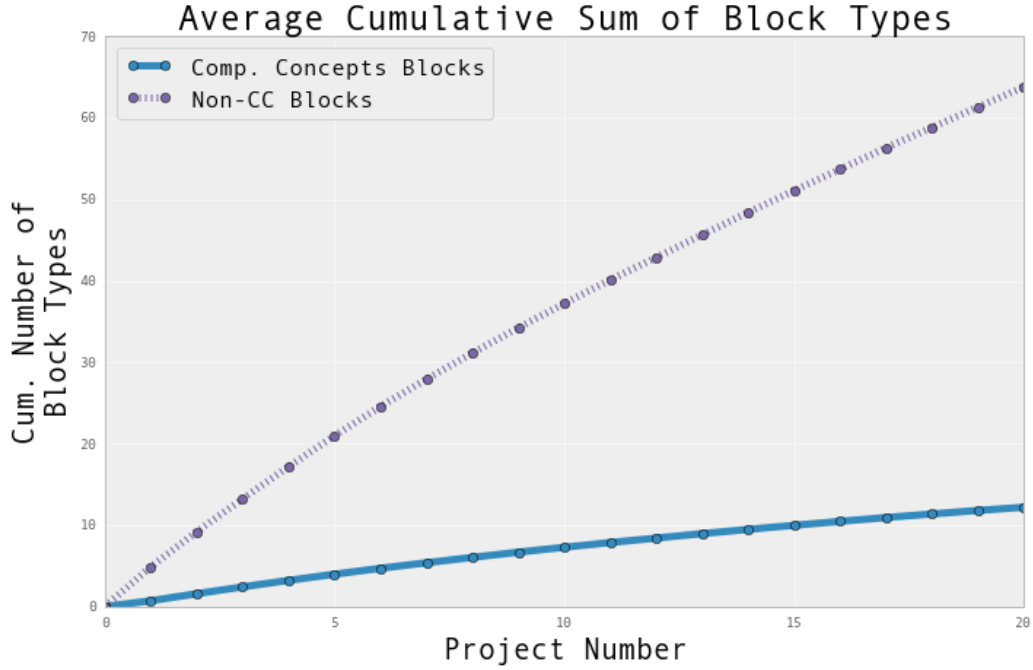


Figure 3-2: Cumulative number of new blocks introduced at a given project

putational concepts follow each other closely. This is significant because it suggests that as users demonstrate domain-specific App Inventor skills, they also demonstrate generalizable skills using computational concepts. We note that the normalized CC trajectory initially lags below the non-CC trajectory for the first 9 projects, perhaps reflecting a period where users focus on familiarizing themselves with the App Inventor environment.

The normalized rate of introducing new block types to projects (averaged across all users) is shown in Figure 3-4. As users create more projects, they introduce fewer blocks (CC and non-CC) to their vocabulary, as evidenced by the decreasing rate of introduction. The rate of CC blocks introduction appears to decrease more substantially than non-CC blocks in the later projects (after 15). This could suggest that there may be a "saturation point" where a user's breadth of skill using CC blocks has encompassed roughly all of the computational concepts in App Inventor or that only subset of CC blocks is sufficient to support the use of a rich set of components and app functionality.

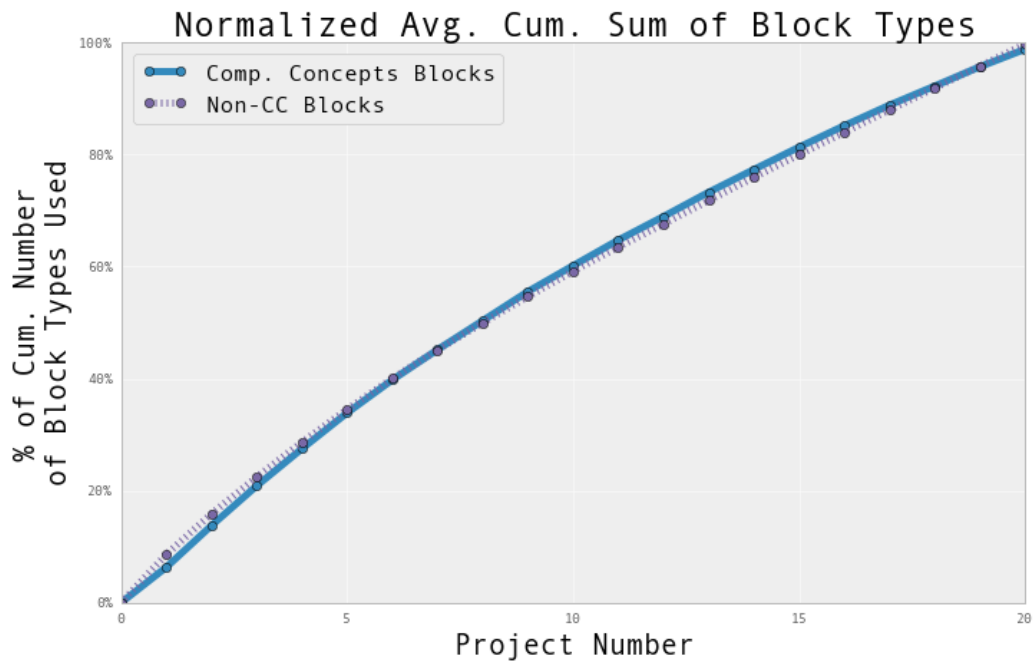


Figure 3-3: Normalized cumulative number of new blocks introduced at a given project

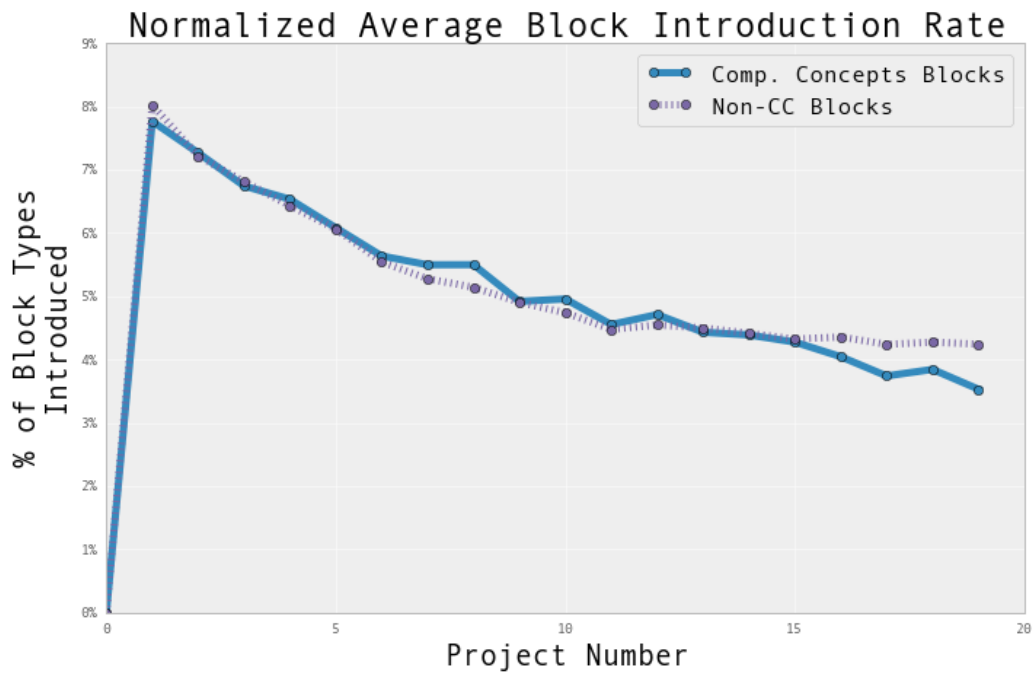


Figure 3-4: Normalized average rate of introducing new block types to projects

In summary:

- The breadth of capabilities to use App Inventor functionality (non-CC blocks) and to use computational concepts (CC blocks) follow each other very closely in general (Figure 3-3). This suggests as people use a broader range of App Inventor functionality, they are demonstrating usage of more computational concepts.
- The demonstrated breadth of capability grows at a decreasing rate as users create more projects (Figure 3-4). So as people develop more projects, they tend to reuse previously used block types rather than introduce new block types to their vocabulary.

3.3 Depth of capability continues to increase

I now analyze the depth of capability or the mastery of certain features and functions. My primary metric for depth of capability is the total number of block types used in a given project, but I also consider the number of events responded to.

Figure 3-5 shows the number of block types in each project, averaged over all 10,571 users. The first five projects likely reflect a period of familiarizing with App Inventor. After the first five projects, the increase in depth of capability is increasing in a somewhat linear fashion for both CC and non-CC blocks. This is also true for the average number of events handled in each project (Figure 3-6).

In general, we see an increase in depth of capability as users create more project. So as users create more projects with App Inventor, they tend to make sophisticated apps that utilize more blocks to enable more robust functionality.

3.4 Clustering to identify CC usage patterns

I cluster users and analyze the three users nearest to the centroid of each cluster to identify common patterns of using CC blocks and differences between the common patterns.

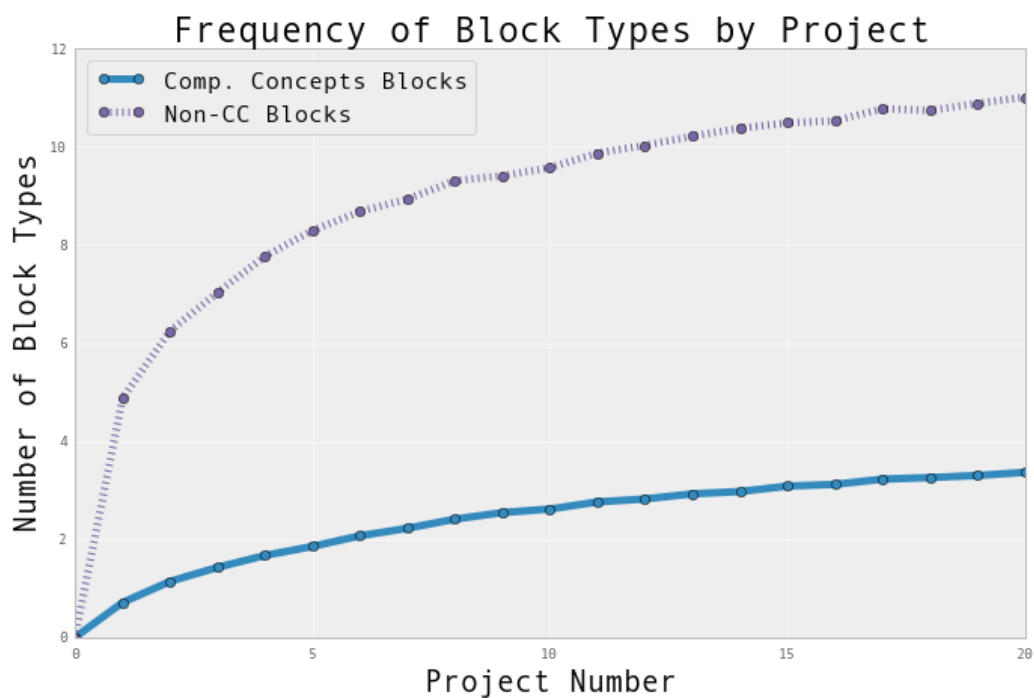


Figure 3-5: Average number of block types used in each project (to measure depth of capability)

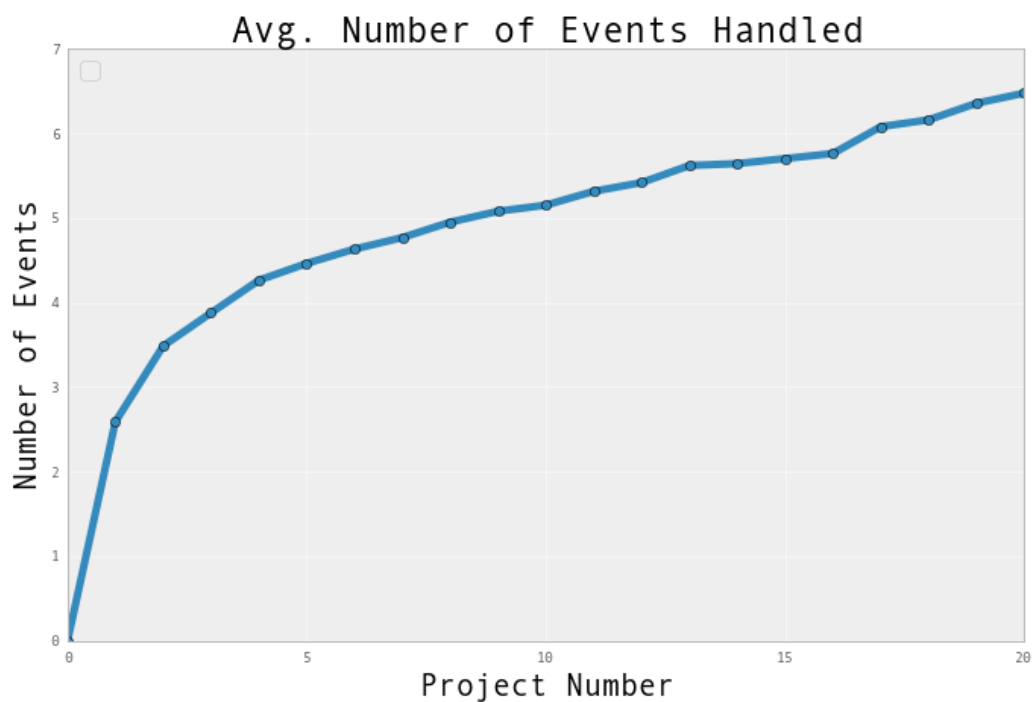


Figure 3-6: Average number of events responded to in each project (to measure depth of capability)

Figure 3-7 shows the results of this elbow method. I select $k = 2$ to be the optimal number of clusters because there is a discontinuous "elbow" at $k = 2$ that shows a significant decrease in the average distance between points in a cluster and the centroid of the cluster for all clusters. The feature vector is the sum of the CC blocks for each project, where the CC blocks are weighted by inverse document frequency (see section 2.5 for IDF weighting). The result is an $n \times d$ matrix where n is the number of users (10,571) and d is the number of projects considered for each user (20). The value at i -th row and j -th column is the cumulative sum of weighted CC blocks used in the j -th project of user i .

Given the feature vector and number of clusters defined, I perform K-Means clustering to cluster users who use CC blocks in a similar pattern. I then analyze the three users closest to the centroid of each cluster and say these users represent the cluster as a whole.

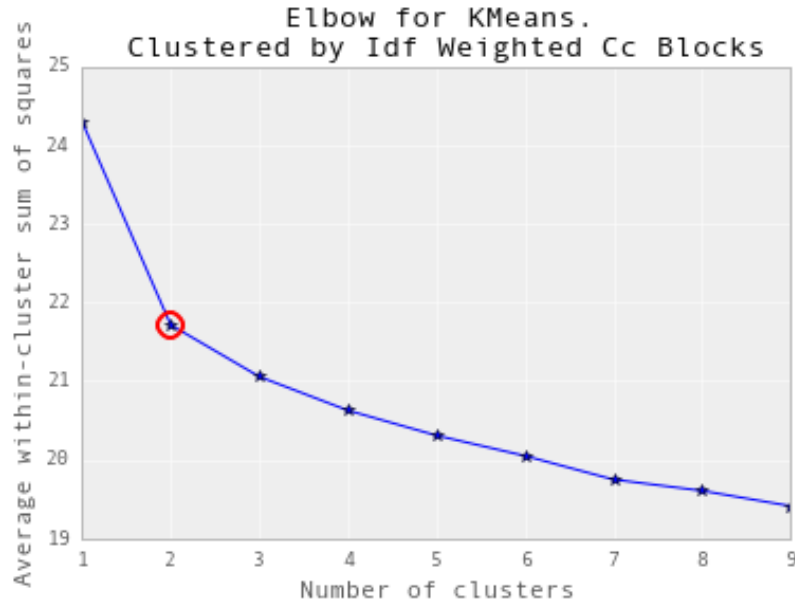


Figure 3-7: Selecting number of clusters by considering average within-cluster sum squared error. $k = 2$ is selected.

Table 3.1 shows information on the clustering. Cluster 0 includes 72% of the users but is a tighter cluster as the average distance from the centroid is less. The representative users are also closer to the center. Cluster 1 includes fewer users but

is a larger cluster as the points are farther apart. The representative users are also farther from the center of the cluster.

Table 3.1: Information on Clusters

Cluster	Num. Users	Avg. Dist. from Centroid	Range of Dist. from Centroid	Rep. Users' Dist. from Centroid
0	7,655	23.8	3.4 - 169.1	3.4, 3.7, 4.0
1	2,916	32.2	9.4 - 155.2	9.4, 9.4, 10.1

We compare the average cumulative IDF-weighted CC block types for each cluster in Figure 3-8. We find that users in cluster 1 maintain a higher cumulative weighted CC blocks vocabulary, suggesting that members of cluster 1 tend to use more CC blocks or more sophisticated blocks than members of cluster 0. The rate of introducing new CC blocks decreases more substantially for cluster 1 as more projects are created, while the rate of introduction for cluster 0 remains more linear.

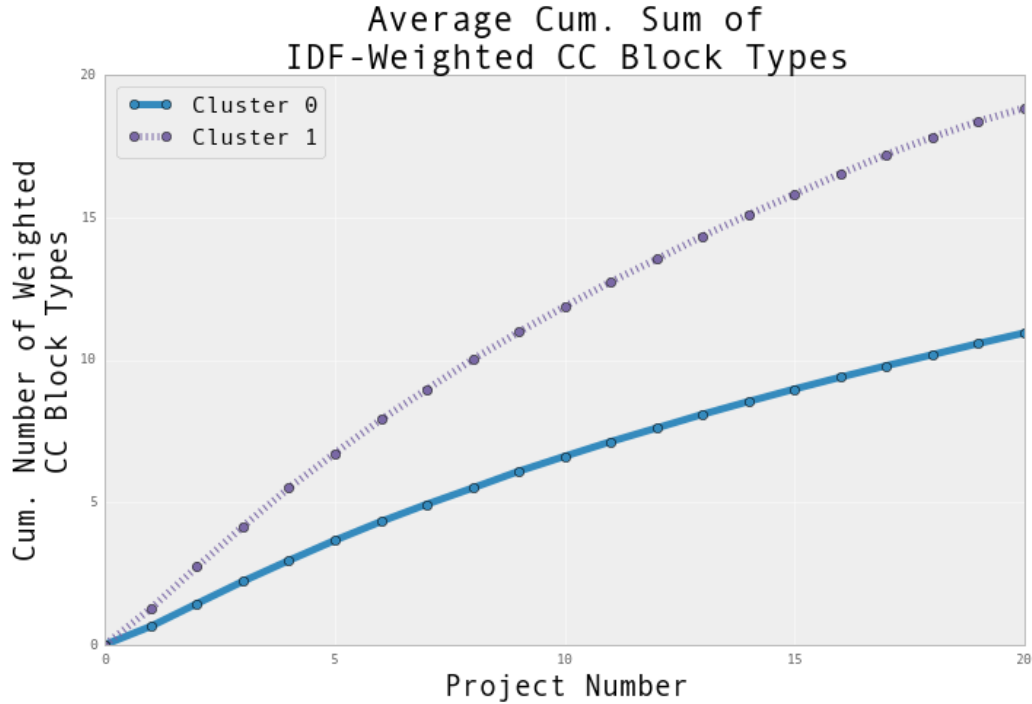


Figure 3-8: Average cumulative sum of IDF-weighted CC block types for each cluster

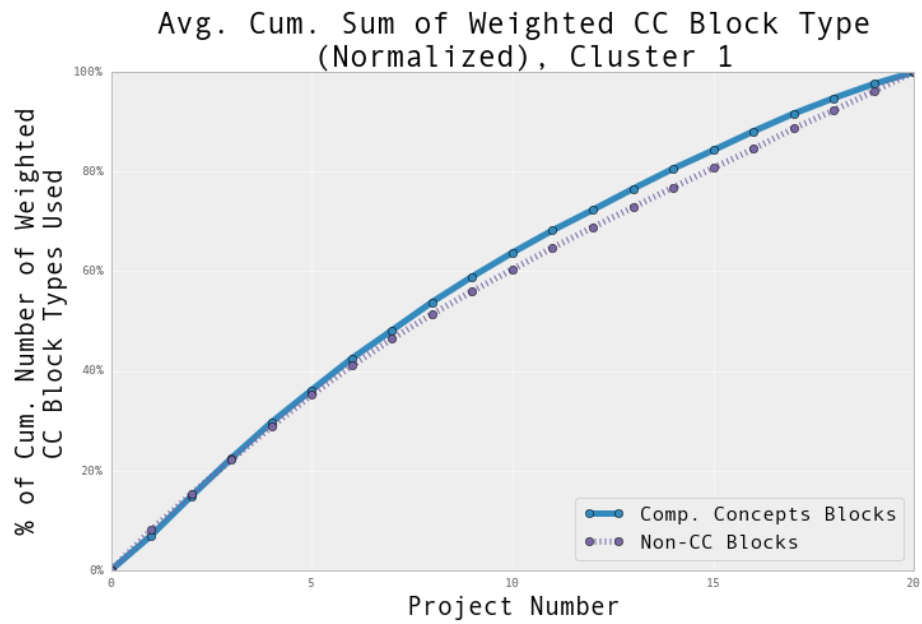
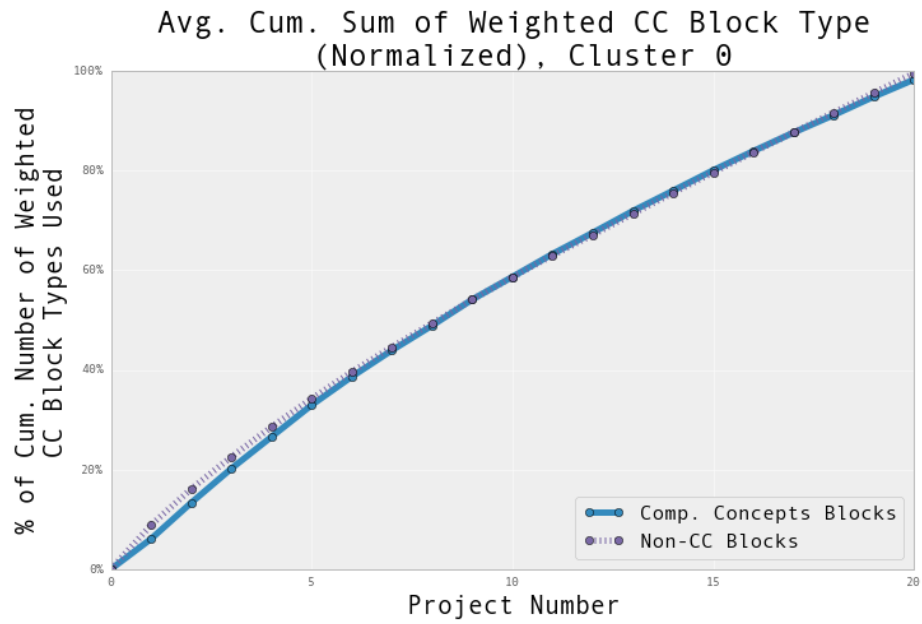


Figure 3-9: Normalized cumulative sum of weighted CC block types for each cluster of users

3.4.1 Representative Users

We say that the three users closest to the center of each cluster are representative users of the cluster. We analyze the order in which the representative users use CC blocks to understand common patterns and identify difference between clusters and across clusters. Information on the representative users is shown in Table 3.2 for cluster 0 and Table 3.3 for cluster 1.

Representative users in cluster 0 did not use a single CC block in the first 4 projects, creating apps that only use basic user interface components (button, textbox, etc.) to perform tasks such as add or subtract numbers. One representative user followed only the first tutorial (Talk to Me), while the other two did not recreate any tutorials in their first 4 projects. This paired with the knowledge that cluster 0 uses fewer CC blocks may suggest that learners benefit from at least some direction via guided tutorials to learn computational concepts with App Inventor. In comparison, each representative user from cluster 1 followed multiple tutorials in their first 4 projects.

From analyzing the representative users in each cluster, it is difficult to identify differences between the clusters, but there are patterns that we identify from considering representative users from both clusters together. Patterns identified for each computational concept type:

- *Variables*: Users tend to access component properties with the `lexical_variable_get` block before using that same block to access global variables.
- *Conditionals*: If statements (`controls_if`) tend to be used early on for all users. These conditionals often check the state or properties of certain components (e.g. *If* a textbox has a valid phone number, send a text message to that phone number)
- *Logic*: Logic/operator blocks tend to be introduced after conditionals. Using logic blocks enable users to check for more states with a conditional (e.g. if textbox has a valid phone number *and* another textbox has a message, send the

text)

- *List*: Lists can be created (`lists_create_with`) and utilized without any functionality to manipulate the list. They are typically displayed with a ListViewer component and represent a predefined static list of items. (e.g. display a list of phone numbers of members of a household)
- *Loops*: Iterators are not often used. This is likely because App Inventor's event driven environment limits the uses for an iterator.
- *Procedures*: Procedures without returns are almost always used before procedures with return values. This suggests the common trend of using procedures to manipulate component properties or provide components with functionality (e.g. writing a procedure to clear a textbox)

Table 3.2: Representative Users' Orders of Acquiring Computational Concepts, Cluster 0

Project Num.	User 1	User 2	User 3
1			
2			
3			
4			
5	controls_if, lexical_variable_set, logic_operation, global_declaration, lexical_variable_get	controls_if	lexical_variable_set, global_declaration, lexical_variable_get
6	controls_forrage		
7	controls_while	logic_false, logic_boolean	controls_if
8	lists_create_with, lists_replace_item, lists_add_items, lists_select_item, controls_foreach		controls_choose
9	lists_remove_item	lexical_variable_get	
10			controls_forrage
11		lexical_variable_set, global_declaration	
12		procedures_defnoreturn, procedures_callnoreturn	controls_while
13			
14			
15			
16		procedures_defreturn, procedures_callreturn, local_declaration_expression	
17		lists_create_with, controls_foreach	
18			logic_boolean
19		lists_pick_random_item	
20		lists_length, lists_select_item	logic_compare

Table 3.3: Representative Users' Orders of Acquiring Computational Concepts, Cluster 1

Project Num.	User 1	User 2	User 3
1			lexical_variable_get
2			
3	controls_if	lexical_variable_get, logic_boolean	
4			
5			
6			
7	global_declaration, lexical_variable_get		
8		controls_if	
9			
10			
11			
12	lists_create_with, logic_boolean	logic_false, logic_or	logic_boolean
13			
14	lexical_variable_set		controls_if, global_declaration, logic_compare
15		procedures_callnoreturn	lexical_variable_set
16			
17	lists_pick_random_item, logic_false		
18		procedures_defreturn, lists_create_with, lists_copy, procedures_callreturn, procedures_defnoreturn, lexical_variable_set, global_declaration, lists_is_in, lists_append_list, controls_foreach	
19	logic_compare		logic_false
20	procedures_defnoreturn, procedures_callnoreturn		

Chapter 4

Discussion

In this section, I propose a behavior of "breadth before depth" where users tend to familiarize themselves with a wide array of components and block types in earlier projects and then develop a mastery of previously learned skills in later projects (section 4.1). I hypothesize that App Inventor users' depth of capability continually increases over time because App Inventor is a robust and extensible environment so it is still engaging to advanced and long-term users. I discuss the results in the context of other blocks-based environments (section 4.2) and text-based environments (section 4.3). I note that connections between programming blocks differentiate statements (code that produces an action) from expressions (code that produces a result). This makes learning to program with App Inventor unique when compared to learning to program with a text-based programming language such as Java. I propose a 3-phased progression of complexity for computational concepts that is based on how users interact with components (access component properties, use component methods, change component state). I also note limitations 4.4 and future work 4.5.

4.1 Developing breadth before developing depth of skill

Our analysis suggests that users begin by developing their breadth of skill and then go on to develop their depth of skill in later projects. That is, users will learn to use different components and different blocks in their earlier projects, and then reuse previously used concepts in more advanced ways in later projects. The decreasing rate of new block types being introduced into projects and the increasing number of block types used in later projects supports this claim. The transition from learning new skills to developing previously used skills is continuous but it appears that after creating 8-10 projects, users typically focus less on acquiring new skills and begin to focus on using developing previously used skills to create more sophisticated apps and use computational concepts again. This reuse is necessary for learning.

Reusing a concept multiple times is necessary to actually learn a concept, so there is a need to create many projects (or iterate on a project many times) to learn generalizable computational concepts with App Inventor. So, to actually learn computational concepts from App Inventor, two things are required: Learners must continue using App Inventor for a long enough time to transition to developing the depth of their capabilities and the environment must be extensible enough such that there are more complex and sophisticated artifacts to create.

I select users who created at least 20 projects, which accounts for only the top 1.4% of users. Previous analysis of App Inventor found that less than 20% of users created more than 4 projects [7]. Our analysis suggests that a typical user typically does not create enough projects with App Inventor to develop mastery of skills (develop depth of capability). This lack of user retention is typical in open programming environments, as I will elaborate on in section 4.2, because a significant portion of users are self-directed and without an instructor. For those who use App Inventor in formal learning environments, this lack of retention is less of an issue.

App Inventor is extensible enough for use in formal environments and long-term curricula. A suitable learning environment must follow a "low-floor, high-ceiling"

design in that it must be usable enough such that beginners can easily create a basic yet functioning program (low floor), but also have extensible capabilities such that advanced users can also benefit (high ceiling) [8]. Previous work with App Inventor has found that the environment is extensible enough for advanced users because it enables users to create apps that connect to the external world [30]. That is, advanced apps connect to sensors on the phone, the internet (e.g. HTTP requests), and physical components such as Arduino via Bluetooth connection. This functionality for App Inventor suggests that App Inventor has a "high ceiling" for more advanced users to benefit from it and create enough apps such that they can develop the depth of their capabilities to use computational concepts. App Inventor has enough advanced functionality such that it can remain engaging for long-term users and can therefore be integrated with a long-term formal curriculum.

4.2 Comparing to Scratch

I compare my analysis on measuring the progression of sophistication with the work by Christopher Scaffidi on the progression of sophistication of Scratch projects [19]. I find that both Scratch and App Inventor have a plateauing in the breadth of demonstrated capability. That is, users only learn a subset of features available on each platform. From there, Scaffidi found that the depth of capability for Scratch projects actually decreased over time, whereas we find that App Inventor users' depth of capability increases over time as they tend to make more sophisticated projects. Scaffidi attributes this decrease in depth of capability over time in Scratch to user retention problems; advanced users tend not to stay with Scratch.

App Inventor offers more functionality and perhaps seems more of an authentic programming experience when compared to Scratch. Whereas Scratch is primarily designed for 8 to 16 years old, App Inventor has proven useful to grammar school students as well as college aged students and even industry professionals who do not have a strong programming background (known as end-user programmers; see section 4.4.2). This is likely because Scratch projects can only be shared within

the Scratch environment while App Inventor enables users to create fully functional Android applications that they can use, share, and even put on the Google Play store. Since blocks-based programming tends to have a perception of inauthenticity when compared to text-based programming (see section A.3), creating apps that a wider population of people (not just App Inventor users) will find useful likely helps attribute to the authenticity of App Inventor which helps retain users. Nevertheless, user retention is a challenge in open programming environments like App Inventor.

User retention is a ubiquitous challenge to open, online environments such as App Inventor. Similar open programming environments point to user drop-off before users develop their depth of capability. Research with Scratch found that breadth and depth of capability decreased as time progressed, likely because more advanced users stopped using the environment [19]. Research on Microsoft TouchDevelop, an environment that enables the programming of apps from a mobile device, found that over 70% of users learned a few features initially then stopped learning new features, suggesting that the TouchDevelop users also stop focusing on developing the breadth of their capability at some point [12]. Because these online environments are so accessible with their easy sign-up process and intuitive interface, retaining users will always be a challenge to environments such as App Inventor. Nevertheless, there exists a need to develop a service that is sophisticated enough to still be engaging for users with previous programming experience or long-term users.

4.3 Learning programming with blocks in App Inventor \neq learning with text languages

Blocks programming languages are not text programming languages. Likewise, learning programming with blocks languages is not the same as learning programming with text languages. The order and progression of using computational concepts with blocks programming in App Inventor deviates from what we might expect when teaching with text languages. This is likely because blocks languages discretize con-

cepts in separate blocks and because of App Inventor’s event-driven programming environment. I use my analysis of the progression of representative users (see section 3.4.1) to compare programming with blocks in App Inventor to programming with Java, although this analysis should generalize to other text-based programming languages. Whereas previous work has analyzed differences in perceptions between blocks-based and text-based languages, I consider differences based on usage patterns (See section A.3 for related work on differences in perceptions of blocks and text languages.).

4.3.1 The connections in block languages separate statements from expressions

The Blockly language in App Inventor has two distinct connections for *statements* and *expressions*. Statements produce an action and blocks are added vertically. Expressions produce a resulting value and blocks are added horizontally. These different connections create visual cues which help novices differentiate between producing actions with statements and producing values with expressions [4]. In Figure 4-1, the top procedure (`increment_counter`) contains a statement which increments a counter label on the app and does not return anything. The blocks are added to this procedure vertically. The bottom procedure (`square_values`) contains an expression and returns the squared value of the input parameter. The blocks in the procedure that returns the squared value are added horizontally because the procedure contains an expression and returns a value.

Blocks languages discretize concepts that otherwise may seem connected or atomic in text languages. In Java, determining whether a method returns a value typically requires at most changing the method header and adding a return statement. We associate methods as a concept and a return statement (or lack thereof) as an attribute of a method. In App Inventor, procedures with and without return values are entirely different blocks, as shown in Figure 4-1. We find that only 15% of procedures return a value in App Inventor. As mentioned in section 4.3, this is likely because the App

Inventor environment lends itself to using procedures to manipulate components and therefore not return anything. But nevertheless, we find that procedures with return values are first used well after procedures without return values are used, if at all. So, blocks languages may separate concepts that would otherwise be seen as connected in Java. This discretization does prove to be necessary for blocks languages since blocks have different connections.

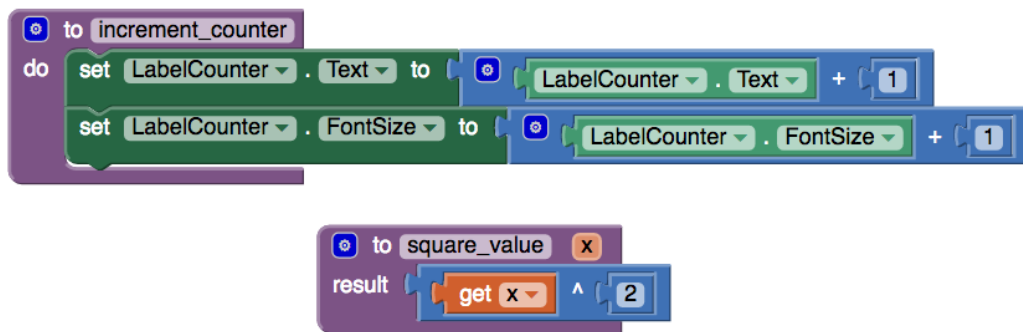


Figure 4-1: Procedures without return values (top) and with return values (bottom).

Separating concepts that appear atomic in text languages is necessary because of the different connections between blocks. Figure 4-2 shows two different if/else blocks in App Inventor. The left one determines which statement to execute, whereas the right one chooses which expression to return. In Java, deciding which script to execute and deciding which value to return requires the same if/else statement. The different connections for blocks programming requires multiple blocks to reflect the functionality of one concept in Java. This may limit learners' perceptions of what computational concepts (conditionals in this case) can and cannot do.



Figure 4-2: If/else blocks to determine which statements to execute (left) and which expressions to return (right)

4.3.2 Computational concepts may develop from manipulating components

My analysis has focused strictly on analyzing computational concept blocks that are agnostic of which components are used in the app. In my analysis, I treat all component-related blocks as separate from computational concepts (as non-CC blocks). From analyzing representative users more closely, I find that using and manipulating component properties may reflect using computational concepts. I provide an example of using the block to get a variable (`lexical_variable_get`) to access a method parameter as well as a global variable. This block is equivalent to both variables and method parameters in Java.

An interesting observation is that users often use the `lexical_variable_get` before they use the blocks to define or set variables. This is because the `lexical_variable_get` block can access both a variable as well as component-specific parameters. Figure 4-3 shows an example of this, as the `lexical_variable_get` blocks (in orange) access the coordinates the canvas component was touched at as well as the value stored in the global variable `dotsize`. So, `lexical_variable_get` is a single block that is used to both access component parameters as well as access variables. In this case, the blocks-based language has a single block that is overloaded and reflects multiple concepts in Java.

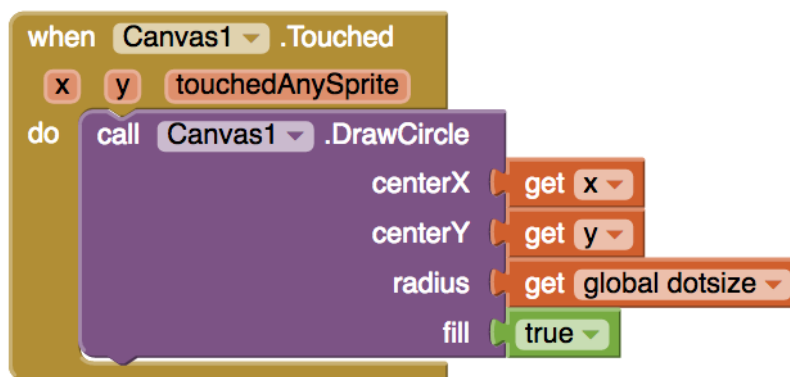


Figure 4-3: The `lexical_variable_get` block (in orange) can access both component parameters (`x`, `y`) as well as global variables (`dotsize`)

4.3.3 Measuring complexity of computational concepts relative to App Inventor's event-based model

I analyze the order in which learners use computational concepts to determine the perceived complexity of using different concepts. I say that users perceive CC blocks that are used less frequently (as shown in Figure 3-1) or introduced in later projects (Tables 3.2, 3.3) as more complex. I determine the complexity of concepts relative to how users interact with the components of their apps. Learners transition from accessing and responding to component events, to manipulating component functionality, to setting component properties and states. I detail the computational concepts used in each phase of complexity:

In the first phase, users access component properties and respond to component events. In this phase, users tend to get component properties, use conditionals to decide between statements to execute, and use logic blocks to make more advanced conditional cases. Typical behavior in this phase:

- The get variable block (`lexical_variable_get`) is used to access component properties (e.g. the value in a textbox or location of a screen touch).
- Conditional statements (`controls_if`) are used to decide which statement to execute based on the state of a component (e.g. if a checkbox is checked).
- Logic blocks (e.g. `logic_boolean`, `logic_compare`) are used to make more advanced conditional cases.

In the second phase, users manipulate component functionality. In this phase, users typically call component methods. Users tend to define and set variables, define procedures without returns, create lists, and use basic loops/iterators. Typical behavior in this phase:

- Global variables are defined and set (`global_declaration`, `lexical_variable_set`). In the previous phase, learners utilized the get variable block to access component properties. Now, learners use it to access global variables they define.
- Procedures without returns are defined and set (`procedures_defnoreturn`,

`procedures_callnoreturn`). Users typically define these procedures to replicate similar functionality across multiple components (e.g. moving an image sprite in a different direction depending on which button is pressed)

- Lists are created but often not manipulated (`lists_create_with`). Users can create a list with predefined values to display information in a List Viewer component or they can create a color by specifying a list of RGB values. Information in this list is often never manipulated in this phase.
- Basic loops (`controls_forRange`, `controls_forEach`) may be used. As mentioned previously, App Inventor's event-based model tends not to lend itself to require iteration, so iterators are introduced later than one might expect when learning a text-based language.
- Logic blocks are used to make more advanced conditional cases.

In the third phase, users tend to change components' properties and states. Users tend to define procedures that return values, manipulate lists, and use iterators. Typical behavior in this phase:

- Procedures with return values are defined and called (`procedures_defreturn`, `procedures_callreturn`). Example uses for procedures with return values include determining user-defined states (e.g. if a sprite is growing or shrinking) and making mathematical calculations (e.g. determining distance travelled with a location sensor).
- List operations (`lists_pick_random_item`, `lists_select_item`, `lists_append_list`) and list properties (`lists_length`, `lists_is_in`) are utilized. Examples uses include keeping track of sprites that appear and disappear in a game or selecting a random output in a magic 8 ball app.
- While Loops tend to be used to iterate based on the state of a component (e.g. number defined on a slider) or a global variable. These loops tend to require more sophistication because the conditional to continue iterating must be defined and the user must increment some counter in the while loop to prevent an infinite loop. In the other iterators (`controls_forRange`, `controls_forEach`), this incrementation is built in.

This three phase description of perceived complexity in App Inventor provides insight into the order users tend to use computational concepts and therefore the order of the perceived complexity of the concepts. This information may prove useful when determining the order concepts should be introduced in a curriculum taught with App Inventor. While I base this information off users near the centers of the clusters, these findings are more qualitative than other analysis in this thesis. I also ignore differences in users and learning environments. Conceivably, users with prior experience with a text-based programming language or Scratch may use computational concepts differently than a user with no prior programming experience. Likewise, the order and perceived complexity of concepts may differ in a classroom environment with a trained instructor present when compared to learning App Inventor independently and without direct guidance. I consider these factors as opportunities for further research (section 4.5).

4.4 Limitations

4.4.1 Measuring blocks is not enough

As mentioned in section 4.3.2, generalizable knowledge is not limited to using computational concepts. Skills necessary in other programming languages such as accessing and manipulating object (component) properties and states is done with component specific blocks which by definition are non-CC blocks. So measuring only CC blocks does not sufficiently encompass the computational thinking skills that users can learn from using App Inventor.

4.4.2 End-user programmers care less about computational thinking

Many of App Inventor's users can be categorized as *end-user programmers*. End-user programming can be defined as "programming to achieve the result of a program primarily for personal, rather than public use" (Ko 2011, [10]). End-user programmers

write programs to support some goal within their domain of expertise. Examples include doctors from in India using App Inventor to create an app to spread awareness and treatment options for diabetes [18], and young people in Oakland, CA using App Inventor to create a mobile game that teaches people about how to save water during the drought [32]. These are examples of people that use App Inventor to write programs that support their domain-specific goals.

These users are not as interested in developing computational thinking skills as they are in developing skills to create apps to accomplish their specific tasks. So while this thesis focuses primarily on computational thinking skills, it is still important for users to develop their knowledge of App Inventor skills.

4.5 Future Work: Data on project progression, users would extend work

Looking at the development of specific projects and considering different types of users. would provide more insight into how users develop their programming and computational thinking skills.

While this thesis focuses on the progression of a user across projects, in-depth analysis of the development of projects would shed insight into learner tendencies and behaviors. For this analysis, it would be ideal to analyze a learner's step-by-step process as they develop an app and see how this process changes as they create more apps. Logging the app-development process would provide more information than showing the final state of the app (the data I analyzed for my thesis) because it would provide insight into the learner's programming behaviors, patterns, and mistakes. Interesting research directions include how users develop iteratively in App Inventor's blocks-based environment or how users debug or program through trial-and-error. Work by Weintrop 2015 suggests that high school students perceived that blocks-based languages lent themselves to trial-and-error programming [27]. Blikstein 2011 is an initial step towards developing metrics to identifying patterns to students'

programming habits [2].

Considering how different users create apps and learn with App Inventor would be a further extension of this thesis. Users could be categorized by age, prior experience (previous computer science courses, previously used Scratch, participated in hour of code, never coded before), objectives (learn programming, build apps), or environment they use App Inventor in (in-person class, online class, self-learning). By understanding how usage patterns differ by different types of users, we would be able to personalize curriculum and learning resources to different types of users.

Another extension to this work would be to analyze particular blocks that pertain to abstraction, such as procedures and variables. A limitation to this analysis is that when measuring development of sophistication, all blocks are treated equally. As explained in section 2.2, IDF weighting (used to weight blocks in previous analysis for Scratch) is not appropriate for the blocks because some blocks relate to rarely used components or rarely used component functionality. An alternative approach would be to follow the use of particular blocks across projects. Analyzing the number of variables and procedures defined and the number of times each variable or procedure was called may provide further insight into how users' sophistication with the concept of abstraction develops.

Chapter 5

Conclusions

I conclude by noting the implications of this research for computer science teachers, education researchers, and App Inventor students. I then list the contributions of my thesis.

5.1 Implications

5.1.1 Teachers can develop curriculum with App Inventor's event-based environment in mind

This work makes us more able to develop a curriculum that teaches computer science principles and computational thinking using App Inventor *with App Inventor in mind*. By recognizing App Inventor's blocks-based programming language and events-based model, teachers are able to develop a curriculum that leverages App Inventor. In section 4.3.3, I define 3 phases that provide an order of increasing complexity for computational concepts in App Inventor. With this concept complexity measure, curriculum can start with what comes naturally in App Inventor for beginners, not what comes naturally for a text-based programming language or another environment. Furthermore, the finding that learners develop breadth before depth of capability (section 4.1) suggests that teachers should develop a curriculum that involves creating at least 10 projects to develop learners' capability to use a wide variety of blocks and

more than 10 projects to develop mastery of previously used skills.

5.1.2 Researchers can quantitatively measure progression of skill in blocks-based environments

This work quantitatively measures the sophistication of skill demonstrated by users against two dimensions: breadth and depth. This connects previous work in measuring sophistication and measuring learning trajectories (breadth of capability) and shows that these quantitative techniques of measurement extend beyond Scratch and are applicable to App Inventor. So, I identify techniques of measuring the progression of skill at scale and suggest that these techniques are generalizable to other blocks-based programming environments.

This work also validates an assumption made by previous researchers ([19], [30], [31]) that users tend to follow a similar pattern in learning generalizable computational concepts as they do in learning domain-specific functionality. So, we are able to consider all blocks when measuring sophistication, even if most blocks in App Inventor do not relate to knowledge that generalizes across different programming domains.

5.1.3 App Inventor learners can measure their progression of learning

These findings are early steps toward enabling App Inventor learners to monitor their own learning and progression. Most App Inventor users create apps outside of classroom or clubhouse environments, so they do not have an expert providing them guidance. Knowing the types of computational concepts that App Inventor teaches and having an order of increasing complexity for these computational concepts (as defined in section 4.3.3) could enable students to keep track of their own learning as they learn to program with App Inventor.

To help guide users' learning as they create apps, we might imagine a map or guide that directs users to build apps that include computational concepts of increasing complexity or a tool that analyzes a user's App Inventor portfolio and notes what

skills they have and have not used and recommends a relevant tutorial or learning resource to increase the breadth or depth of capability for a given user.

5.1.4 Contributions

The big idea behind this thesis is that we can quantitatively analyze the progression of using computational concept skills and model how users become more sophisticated with using these skills which generalize to other programming domains. By understanding what users are learning by creating apps with App Inventor, we take a step towards the long-term objective of connecting the knowledge acquired by open-ended learning with what is being taught in formal classrooms.

Another important contribution of this thesis is the clustering of users who share similar learning patterns with learning computational concepts and investigating users who are representative of the larger population. With this, we can better understand the perceived complexity of concepts by investigating the order in which learners use computational concepts. This work can guide future curricula that uses App Inventor as well as provide useful insight for adaptive tutors that can guide future App Inventor users and create more personalized learning experiences.

In conclusion, for this thesis I:

- Modeled the demonstrated breadth and depth of App Inventor user capability quantitatively. Breadth is modeled as a learning trajectory, or cumulative number of new block types used at each of a given user's projects. Depth is measured by the number of unique block types in project and the number of events responded to.
- Compared the development of domain-specific skills to use App Inventor functionality with the development of generalizable skills to use computational concepts.
- Compared results to Scratch and explained differences.
- Identified a common pattern of computational concept usage where learners use new computational concepts in earlier projects (first 10), then reuse previously introduced computational concepts to develop more sophisticated apps.

- Identified differences in learning to program with a blocks-based language in App Inventor's event-based environment and learning with a text-based programming language such as Java.
- Defined a concept complexity measure that separates computational concepts into three phases based on users' developing knowledge of component usage.

Appendix A

Related Work: Computational Thinking Frameworks, Measuring Demonstrated Skill

Scratch is among the most similar environments to MIT App Inventor. It is a visual blocks-based environment used to create media projects (website: [21]). Figure A-1 shows Scratch blocks that make a sprite move, plays music, and ends with the sprite saying something.

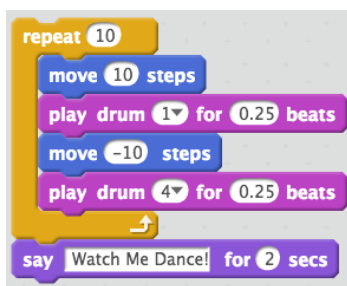


Figure A-1: Blocks from Scratch, an environment similar to MIT App Inventor

Much of the previous work that this thesis builds off of is work done with Scratch. In particular, I adapt the use of a learning trajectory to model informal learning from a proof of concept by Yang 2015 [31] and computational (thinking) concepts from Brennan 2012 [5].

A.1 Breadth and depth are measures of demonstrated skill

Huff 1992 developed a questionnaire to measure the *sophistication* of users in end user computing (EUC) [9]. Three fundamental aspects of EUC were identified:

- ***breadth*** of capability: a broad understanding of knowledge and skill
- ***depth*** of capability: mastery of certain features and functions
- ***finesse***: ability to *creatively* apply EUC

This thesis focuses on measuring the breadth and depth of demonstrated skill. Finesse is out of the scope of this thesis but perhaps an opportunity for future work.

Scaffidi 2012 would use Huff’s model to measure the progression of elementary programming skills in Scratch [19]. Relating to Scratch, Scaffidi grouped similar primitives into different categories. Breadth was the number of distinct categories of primitives used per project. Depth was the total number of primitives invoked in a project.

The contributions of Scaffidi’s work include converting Huff’s model to Scratch such that skill could be measured quantitatively by analyzing project data and without surveying users. Scaffidi concluded that the average depth and breadth of skill Scratch users demonstrated actually *decreased* over time, as shown in Figure A-2. Four possible explanations were proposed: early dropout of more skilled users, data inconsistencies, remixing (building off of other users’ publicly shared projects), and community-wide decrease in complexity of projects.

For this thesis, breadth of capability is modelled by a learning trajectory, as proposed by Yang 2015 [31]. Depth of capability is modelled by considering the number of block types used in projects over time.

A.1.1 Learning trajectories model the *breadth* of capability

The concept of a *learning trajectory* is first introduced by Yang 2015 for Scratch [31] and used by Dasgupta 2016 to empirically verify that Scratch programmers could

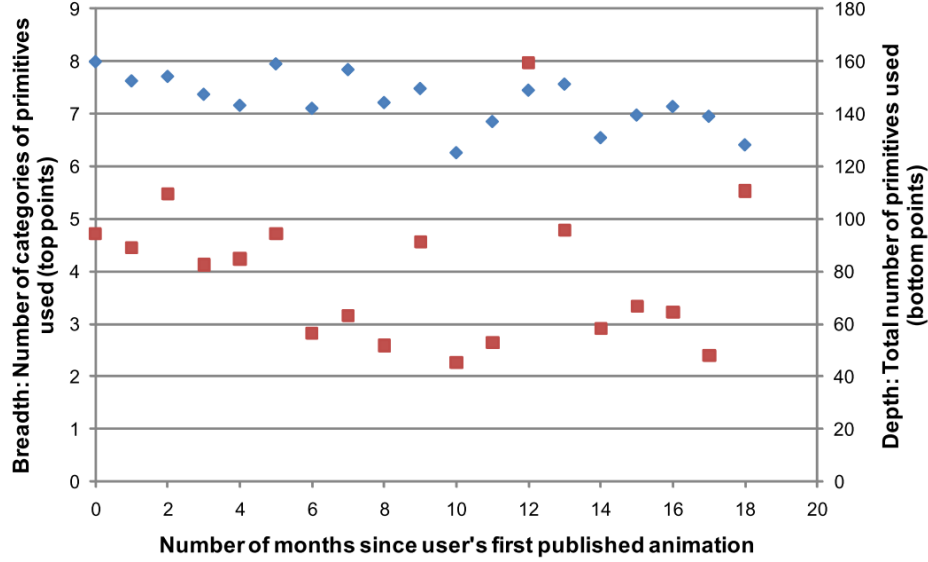


Figure A-2: Results from Scaffidi [19] showing a decrease in average breadth and depth in Scratch projects over time.

increase their programming skills and knowledge of computational thinking concepts through remixing other users' code [6]. In this thesis, I adapt the concept of a learning trajectory for App Inventor and use it to measure the *breadth* of skill to use App Inventor functionality and computational thinking concepts.

Work by Yang 2015 modeled learning trajectories and identified learning patterns at a microscopic (individual user) and macroscopic (cluster) level [31]. Yang measured 3 things: Amount of learning, rate of learning, and potential prior knowledge. Amount of learning is measured by considering the cumulative vocabulary of block use as a user creates more projects over time. The rate of learning is measured by the number of block types used for the first time for each project. The potential prior knowledge is considered by measuring the first value in the trajectory. The contributions of this work: Modeling informal learning as a quantitative trajectory, identifying patterns of learning and corresponding sub-populations.

A.1.2 The number of block types in a project measure the *depth* of capability

My previous work measures the intricacy of App Inventor projects by considering the number of block types in a project (Xie 2015 [30]). This method was found to be more effective than merely counting the number of blocks in a project because counting the number of blocks would bias the intricacy of projects that do not exhibit code reuse (procedures, variables) higher. In other words, a project that copy and pasted identical code in multiple locations should be considered less intricate than a project that used a procedure. I will reuse this idea of counting the number of unique blocks to model the depth of capability.

A.2 Computational Concepts are a dimension of Computational Thinking

Jeannette M. Wing first defined *computational thinking* a decade ago: "Computational thinking involves solving problems, designing systems, and understanding human behavior, by drawing on the concepts fundamental to computer science" [28]. Computational thinking is first and foremost about abstracting and decomposing complex tasks into smaller ones. It can be thought of as the third pillar of science: Theory, experimentation, and computation [29]. Since its first mentioning, much of the research emphasis on computer science education has centered around this ever expanding term that is computational thinking.

Because computational thinking is intended to be useful to anyone, it has a multitude of definitions depending on person and context. In *The Emotion Machine*, Marvin Minsky refers to words that describe the mind, such as (computational) thinking as *suitcase words*. We fill up these suitcase words "with far more stuff than could possibly have just one common cause" [14]. There are various meanings to computational thinking, so the definition described in the context of this thesis does not align perfectly with the definition of computational thinking used in other contexts. I

urge readers to not get too caught up on inconsistencies between the definition in this thesis compared to other work because computational thinking is simply a suitcase word with many meanings packed into it.

We reference computational (thinking) concepts from the Scratch assessment framework from Brennan 2012 [5]. This computational thinking framework consists of three dimensions:

- ***computational concepts***: concepts developers engage with as they program (e.g. conditionals, procedures)
- *computational practices*: practices developers develop as they engage with concepts (e.g. debugging)
- *computational perspectives*: perspectives developers form about the world around them and about themselves (e.g. expressing, connecting)

Analyzing projects that users have created was shown to be effective at assessing computational concepts. So, this thesis focuses on computational concepts present in users' projects.

A.3 Blocks languages are not perceived the same as text languages

Weintrop 2015 performed a study to understand how high school students view blocks-based programming tools, why they were perceived to be easier to use, and how they were different from text-based programming [27]. In particular, Weintrop compared Snap!, an extended reimplementaiton of Scratch which features the ability to create custom blocks, with Java [22].

High school students found blocks programming easier for the following reasons:

- Blocks are easier to read because they appear more like English.
- Blocks provided visual cues such as shape and color
- Blocks are easier to compose and tinker with because have fewer syntactic concerns (compared to text)

- Blocks serve as memory aids because they are organized and students can see them instead of having to recall them (as they would in a text language)

Students identified three differences to blocks-based and text-based programming languages: trial and error programming, pre-fabricated commands, and visual enactment of progress. Students noted how Java was not conducive to trial-and-error programming. They also noted that text-based environments lack the pre-fabricated commands that blocks-based programming environments have. Finally, students found blocks-languages to have greater visual affordances when being executed, a quality that speaks more to the Snap! environment than to blocks programming in general. Figure A-3 shows reported differences between the blocks-based Snap! environment and Java at the mid-point and conclusion of the study.

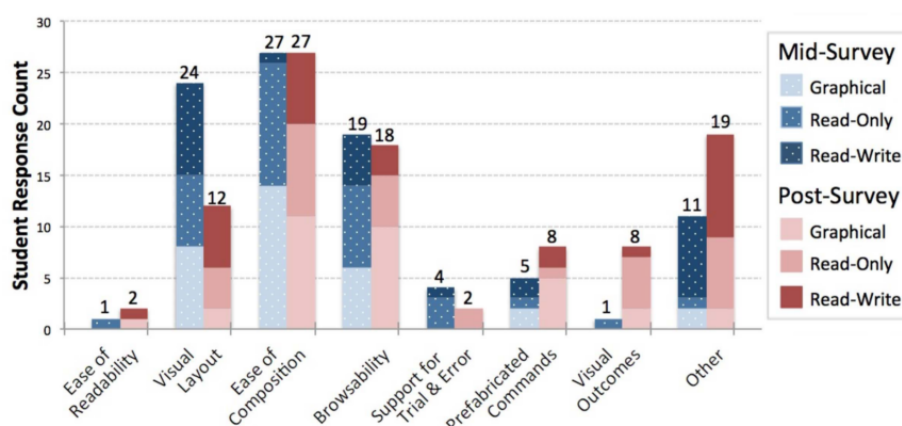


Figure A-3: Student reported differences between Snap! and Java at mid-point and conclusion of study (from [27]).

Students noted several drawbacks to blocks-based programming compare to text-based programming: Less powerful, slower to author and more verbose, and inauthentic. Students perceived that with text-based languages "you can do a lot more" than the limited blocks-based language. Students perceived more possibilities with text-based languages. Furthermore, students found blocks-based environments to be slower to author in that a statement requires multiple blocks compared to "one sentence" in Javascript. It was also noted that blocks languages would be hard to work with for larger projects because the blocks would begin to clutter the screen. Fi-

nally, students perceived blocks-based environments as inauthentic, viewing them as educational and teaching tools rather than "actual code."

Appendix B

Description of Computational Concept (CC) Block Types

Table B.1: Description of Variable Blocks


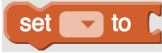



Block Type	Block Image	Description
global_declaration		Define a global variable and assign it a given value.
lexical_variable_set		Set variable to be equal to input.
lexical_variable_get		Returns the value of a given variable.
local_declaration_expression		Create local variable that returns a value (expression).
local_declaration_statement		Create local variable that runs code (statement).

Table B.2: Description of Procedure Blocks

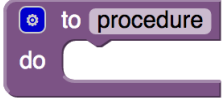
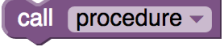
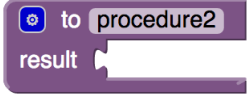
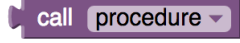
Block Type	Block Image	Description
procedures_defnoreturn		Define a procedure that does not return a value
procedures_callnoreturn		Call a procedure that does not return a value
procedures_defreturn		Define a procedure that returns a value
procedures_callreturn		Call a procedure that returns a value

Table B.3: Description of Loop Blocks

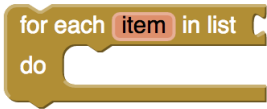
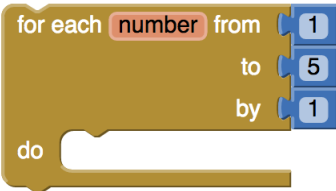
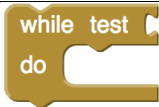
Block Type	Block Image	Description
controls_forEach		Runs the blocks in the 'do' section for each item in the list.
controls_forRange		Runs the blocks in the 'do' section for each numeric value in range from start to finish.
controls_while		Runs the blocks in the 'do' section while the test is true.

Table B.4: Description of Logic Blocks







Block Type	Block Image	Description
logic_negate		Returns true if input is false. Returns false if output is true.
logic_or		Returns true if any input is true
logic_boolean		Returns the boolean true
logic_false		Returns the boolean false
logic_operation		Returns true if all inputs are true.
logic_compare		Tests two things are equal (or not equal)

Table B.5: Description of Conditional Blocks



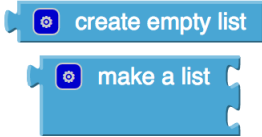
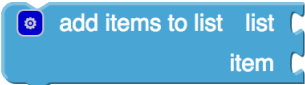
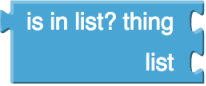




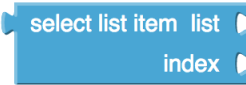
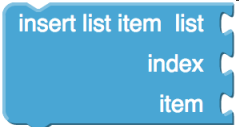
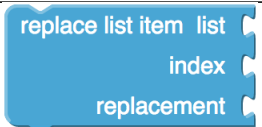
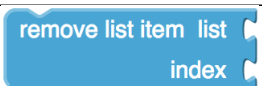
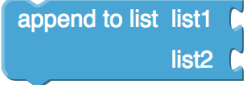






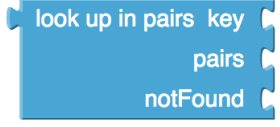
Block Type	Block Image	Description
controls_if		If the condition is true, then execute the 'do' section
controls_choose		If the condition is true, return the result of evaluating the expression for 'then'. Otherwise, execute and return the expression in the 'else' slot.

Table B.6: Description of List Blocks

Block Type	Block Image	Description
<code>lists_create_with</code>		Create new list that is either empty or has items in it
<code>lists_add_items</code>		Add item to list
<code>lists_is_in</code>		Return true if item is in list
<code>lists_length</code>		Return number of items in list
<code>lists_is_empty</code>		Return true if list contains no items
<code>lists_pick_random_item</code>		Return random item in list
<code>lists_position_in</code>		Return index of item (0 if not in list)
<code>lists_select_item</code>		Return item in list at given index
<code>lists_insert_item</code>		Insert item into list at given index
<code>lists_replace_item</code>		Replace item at given index of list
<code>lists_remove_item</code>		Remove item at given index

lists_append_list		Add items in list2 to end of list1
lists_copy		Return a copy of a list
lists_is_list		Return true if input is a list
lists_to_csv_row		Return CSV representation that treats list as a row
lists_to_csv_table		Return CSV representation that treats list as a table
lists_from_csv_row		Given row (in CSV text format), return list where each value in row is an item in the list
lists_from_csv_table		Given text in CSV table format, return list where each item is a list of fields in each row
lists_lookup_in_pairs		Returns the item associated with the key in the list of pairs

Bibliography

- [1] The App Inventor Course-in-a-Box. "<http://www.appinventor.org/content/CourseInABox/Intro/courseinabox>". Accessed April 27, 2016.
- [2] Paulo Blikstein. Using learning analytics to assess students' behavior in open-ended programming tasks. In *Proceedings of the 1st International Conference on Learning Analytics and Knowledge*, LAK '11, 2011.
- [3] Blockly. <https://developers.google.com/blockly/>. Accessed March 14, 2016.
- [4] Blockly Google Group: Statements and expression. <https://groups.google.com/forum/#!topic/blockly/l22CIk5mrxo>. Accessed May 16, 2016.
- [5] Karen Brennan and Mitchel Resnick. New frameworks for studying and assessing the development of computational thinking. In *2012 annual meeting of the American Educational Research Association*, 2012.
- [6] Sayamindu Dasgupta, William Hale, Andres Monroy-Hernandez, and Benjamin Mako Hill. Remixing as a pathway to computational thinking. In *19th ACM Conference on Computer-Supported Cooperative Work and Social Computing (CSCW 2016)*, 2016.
- [7] David Ferreira, John Marshall, Paul O’Gorman, Sara Seager, and Harriet Lau. A preliminary analysis of app inventor blocks programs. In *IEEE Symposium on Visual Languages and Human Centric Computing (VLHCC)*, San Jose, California, Sep. 17 2013.
- [8] Shuchi Grover and Roy Pea. Computational thinking in k–12 a review of the state of the field. *Educational Researcher*, 2013.
- [9] Sid L. Huff, Malcolm C. Munro, and Barbara Marcolin. Modelling and measuring end user sophistication. In *Proceedings of the 1992 ACM SIGCPR Conference on Computer Personnel Research*, SIGCPR '92, 1992.
- [10] Andrew J. Ko, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrance, Henry Lieberman, Brad Myers, Mary Beth Rosson, Gregg Rothermel, Mary Shaw, and Susan Wiedenbeck. The state of the art in end-user software engineering. *ACM Comput. Surv.*, 2011.

- [11] Trupti M. Kodinariya¹ and Prashant R. Makwana. Review on determining number of cluster in k-means clustering. *International Journal of Advance Research in Computer Science and Management Studies*, 2013.
- [12] Sihan Li, Tao Xie, and Nikolai Tillmann. A comprehensive field study of end-user programming on mobile devices. In *Visual Languages and Human-Centric Computing (VL/HCC), 2013 IEEE Symposium on*, pages 43–50. IEEE, 2013.
- [13] J. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*, pages 281–297. University of California Press, 1967.
- [14] Marvin Minsky. *The Emotion Machine: Commonsense Thinking, Artificial Intelligence, and the Future of the Human Mind*. Simon & Schuster, 2007.
- [15] MIT App Inventor. "<http://appinventor.mit.edu/explore/>". Accessed March 14, 2016.
- [16] Code.org. "<https://ram8647.appspot.com/mobileCSP/course>". Accessed April 27, 2016.
- [17] MT513 - Computer Science Principles for High School Teachers. "<https://sites.google.com/a/jcu.edu/mt513>". Accessed April 27, 2016.
- [18] A Diabetic app by Doctors hits the Top Charts in Play Store. "<http://www.pressreleaserocket.net/a-diabetic-app-by-doctors-hits-the-top-charts-in-play-store/438056/>". Accessed April 23, 2016.
- [19] Christopher Scaffidi and Christopher Chambers. Skill progression demonstrated by users in the scratch animation environment. *International Journal of Human-Computer Interaction*, 2012.
- [20] sklearn.cluster.KMeans. "<http://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html>". Accessed April 21, 2016.
- [21] Scratch. "<https://scratch.mit.edu/>". Accessed March 14, 2016.
- [22] Snap! "<http://snap.berkeley.edu/>". Accessed April 25, 2016.
- [23] Karen Sparck Jones. A statistical interpretation of term specificity and its application in retrieval. In *Document Retrieval Systems*, pages 132–142. Taylor Graham Publishing, 1988.
- [24] Reed Stevens and John Bansford. The LIFE Center Lifelong and Lifewide Learning Diagram. <http://life-slc.org/about/citationdetails.html>, 2005. Accessed May 16, 2016.

- [25] Franklyn Turbak, Mark Sherman, Fred Martin, David Wolber, and Shaileen Crawford Pokress. Events-first programming in app inventor. *Journal of Computing Sciences in Colleges*, 2014.
- [26] Tutorials for App Inventor. "<http://appinventor.mit.edu/explore/ai2/tutorials.html>". Accessed March 14, 2016.
- [27] David Weintrop and Uri Wilensky. To block or not to block, that is the question: Students' perceptions of blocks-based programming. In *Proceedings of the 14th International Conference on Interaction Design and Children*, 2015.
- [28] Jeannette M. Wing. Computational thinking. *Communications of the ACM*, 2006.
- [29] Jeannette M Wing. Computational thinking: What and why? *The Magazine of the Carnegie Mellon University School of Computer Science*, 2011.
- [30] Benjamin Xie, Isra Shabir, and Hal Abelson. Measuring the usability and capability of app inventor to create mobile applications. In *Proceedings of the 3rd International Workshop on Programming for Mobile and Touch*, PROMOTO 2015. ACM, 2015.
- [31] Seungwon Yang, Carlotta Domeniconi, Matt Reville, Mack Sweeney, Ben U. Gelman, Chris Beckley, and Aditya Johri. Uncovering trajectories of informal learning in large online communities of creators. In *Proceedings of the Second (2015) ACM Conference on Learning @ Scale*, 2015.
- [32] Youth Radio Releases California Drought Trivia App. "<http://www.eastbayexpress.com/CultureSpyBlog/archives/2015/09/18/youth-radio-releases-california-drought-trivia-app>". Accessed March 14, 2016.