THINKING LIKE A COMPUTER: AN EXPLORATORY STUDY OF INTRODUCTORY PROGRAMMERS' LEARNING PROCESSES IN SCRATCH

by

Minji Kong

A thesis submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Honors Bachelor of Arts in Computer Science with Distinction

Spring 2020

THINKING LIKE A COMPUTER: AN EXPLORATORY STUDY OF INTRODUCTORY PROGRAMMERS' LEARNING PROCESSES IN SCRATCH

by

Minji Kong

I certify that I have read this thesis and that in my opinion it meets the academic and professional standard required by the University as a thesis for the degree of Bachelor of Arts.

Signed: _____
Lori Pollock, Ph.D.
Professor in charge of thesis

Approved: _____
Chrystalla Mouza, Ed.D.
Committee member from the School of Education

Approved: _____
Matthew DeCamp, Ph.D.
Committee member from the Board of Senior Thesis Readers

Approved: _____
Michael Chajes, Ph.D.
Deputy Faculty Director, University Honors Program

ACKNOWLEDGMENTS

First and foremost, I wish to express my deepest gratitude to my advisor, Dr. Lori Pollock. Since offering me the opportunity to participate in CS research, her mentorship, support, and contagious enthusiasm have been invaluable. I would also like to sincerely thank Dr. Chrystalla Mouza for bestowing her trust in me throughout my involvements in the Partner4CS project. I look forward to continuing my academic pursuits under their guidance as I return in the fall, whether it be in-person or remote, to embark on my Ph.D. journey.

I would like to extend my thanks to Dr. Matthew DeCamp for his feedback and advice throughout the thesis process, and Dr. Jean Neff and Dr. Stacy Kotch-Jester for sparing their time and allowing me to conduct the thesis study in their classrooms. In addition, a sincere acknowledgement is dedicated to Dr. Terry Harvey, for providing guidance and helping me find a sense of belonging in CS over the past several years, and Dr. Austin Cory Bart, whose insights and involvement in CS education research have been vital to this thesis process.

Last but not least, I would like to thank my wonderful parents and sister. Without their endless encouragement, patience, and "Kong-gratulating" remarks every step of the way, none of this would have been possible. I'm very grateful to be spending my last undergraduate semester with all of them by my side, especially as we continue to keep ourselves safe and healthy during the COVID-19 pandemic.

TABLE OF CONTENTS

LIST OF TABLES . . . . . **vi**
LIST OF FIGURES . . . . . **vii**
ABSTRACT . . . . . **viii**

Chapter

1 INTRODUCTION AND RELATED WORK **1**

1.1 Introduction 1
1.2 Related Work 3

2 METHODOLOGY AND DATA COLLECTION **6**

2.1 Logging Programming Events 6

2.1.1 The Anatomy of a Log Message 7
2.1.2 Scratch Edit Types 8

2.1.2.1 Implementing the Logging Infrastructure 9

2.2 Mining and Analyzing Programming Processes 11

2.2.1 Preparing Data for Mining 11
2.2.2 Sequential Pattern Mining 12
2.2.3 Manual Analysis of Mined Patterns 14
2.2.4 Deriving All Programming Events Performed 15

2.3 Applying Our Methodology to Study Students' Programming Processes 16

2.3.1 Programming Activity 16
2.3.2 Materials 17
2.3.3 Procedure 18
2.3.4 Evaluating Final Codes 19

3 RESULTS **23**

3.1 Students' Programming Backgrounds 23
3.2 Final Code Performances 23
3.3 RQ1: Anomalous Programming Processes 25

3.3.1 Undirected Programming 27
3.3.2 Misuse of Blocks 29
3.3.3 Communication-driven Focus 30

3.4 RQ2: Costs and Benefits Beyond Final Code Products 30

3.4.1 Benefits Beyond Final Code Products 31
3.4.2 Costs 31

3.5 Threats to Validity 32

4 DISCUSSION AND CONCLUSION **34**

4.1 Discussion and Implications 34
4.2 Conclusions 35
4.3 Enhancing and Implementing Approach in Future Works 35

REFERENCES **37**

Appendix

A STUDENTS' COPY OF THE INSTRUCTIONS FOR THE PROGRAMMING ACTIVITY **40**
B AN OVERVIEW GUIDE TO THE SCRATCH PROGRAMMING ENVIRONMENT **42**
C SURVEY UTILIZED IN EXPLORATORY STUDY **48**

LIST OF TABLES

2.1 Anatomy of example log statements in the Main Event Table 8

3.1 Students' performance on the programming activity solely based on the final code products 24

3.2 Common programming behaviors exhibited during programming activity 26

3.3 Exhibited programming events that appeared in mined patterns but were not considered during clustering 27

3.4 Exhibited programming events that were neither in the mined patterns nor the clusters 28

LIST OF FIGURES

2.1 ProgSnap2-based model of programming process data in Scratch . . 7

2.2 Infrastructure of our Scratch Logging Approach 9

2.3 An example student's sequence of programming events 12

2.4 An example sequential pattern demonstrating repetitive `Move` and `Delete` events with the `Answer` block 14

2.5 An example sequential pattern demonstrating `Move` events with the `Not` operator 15

2.6 CS non-majors completing Scratch Activity 18

2.7 Example solution for the programming activity 20

2.8 Objectives for the first subtask of the programming activity, as demonstrated by an example solution 21

2.9 Objectives for the second subtask of the programming activity, as demonstrated by an example solution 22

3.1 An example sequential pattern exhibiting an undirected move-and-delete approach 28

3.2 An example sequential pattern exhibiting a trial-and-error approach 28

3.3 An example sequential pattern exhibiting the `() and ()` block, which is not needed in the designed programming activity 29

ABSTRACT

As block-based programming environments are increasingly used to introduce students to computational thinking (CT), researchers have developed various techniques and tools for assessing students' CT skills using these visual programming languages. Most approaches focus on analyzing students' final code products; however, insights into how to improve teaching also can be gained from observing students' programming processes as they code, beyond what is learned from their products. Unfortunately, observing each individual student's process firsthand and interviewing students is time-intensive and non-scalable.

This thesis describes a logging and sequential pattern analysis methodology that enables scalable data collection and analysis of students' programming processes in the Scratch visual programming environment. As students code, we record their interactions with the constructs and the environment itself. Using this capability, we explore the kinds of programming behaviors that can be learned from the mined patterns of logs recorded as students performed the same coding task in two college courses introducing programming (37 Education and Human Development majors taking an Ed Tech course and 44 non-CS majors taking a general introductory CS course). Our findings indicate that the logs and patterns can describe one's demonstration of uncertainty with a certain construct, attempted use of blocks that are not required in an assigned programming task, and heavy focus on speech-based constructs that do not involve CT skills, all of which can serve as starting points of discussion concerning how CT teaching methodologies could be improved.

Chapter 1

INTRODUCTION AND RELATED WORK

1.1 Introduction

As the field of Computer Science (CS) continues to drive technological advances in all disciplines, the demand for all students to learn computational thinking (CT) skills is greater than ever. CT, which involves problem solving through thought processes such as abstraction, algorithmic thinking, decomposition, and evaluation [23], is now widely viewed as an essential skill for all students to learn and practice starting at young ages [13].

To provide students an opportunity to learn and practice CT in an engaging and interactive manner, visual programming languages are commonly used in introductory programming classes in both K-12 and university contexts [15]. For many programming students today, their first coding experiences involve such coding languages. Scratch is one of the most popular visual programming tools, enabling its users to create their own fairly sophisticated programs with its graphical programming blocks [21]. The blocks are drag-and-dropped and then snapped together within the programming environment, an approach that a novice programmer is most likely familiar with from building with Lego bricks.

The broader use of such visual programming languages in K-16 educational settings has led to increased interest among researchers in understanding how students apply CT in coding and how their skills could be assessed within these environments. Researchers have developed approaches to evaluating CT competence and programming habits by analyzing users' final codes [18, 1, 16, 22]. While this provides insights into what a user was able to create for a given task, one can only hypothesize about

how they came to that final code. Learning more about the process that the programmer used to create the code could help to better understand their CT process and how common mistakes are made. Learning patterns of introductory programmers' behaviors during coding in these environments can help to refine teaching methods. Unfortunately, observing programmer behavior in-person or interviewing users after they have completed a task is time-consuming and non-scalable.

The goal of this research is to better understand the patterns of novice programming students while they are coding in the Scratch programming environment. We present a logging methodology that captures students' interactions with the Scratch programming environment, as well as a sequential pattern mining approach that takes a set of students' logged interactions as input and extracts anomalous patterns of students' programming processes. By automatically collecting programming event logs and identifying common programming processes among introductory programming students for a given coding task, we can gain insights into interesting behaviors during students' first interactions with CT and Scratch that are not evidenced in their final codes. The automated logging and pattern mining shows promise for large scalable studies.

To explore the promise of this approach, we utilized the automatic logging and mining system in a study of students in two college courses introducing programming using Scratch: 37 Education and Human Development majors taking an Ed Tech course and 44 non-CS majors taking a general introductory CS course. With this set of 81 novice programming students, we designed our study to answer the following research questions:

- **RQ1:** What kinds of anomalous programming processes are we able to expose from logging events from students during their first interactions with CT skills such as abstraction and algorithmic thinking in Scratch?

- **RQ2:** What are the costs and effectiveness of the specific logging events and pattern analysis in providing evidence of programming processes beyond what can be learned from final code products?

Therefore, the contributions of this thesis are (1) the design and implementation of a system to capture and analyze programming event sequences from Scratch coders, as well as (2) the results of a study of novice Scratch programmers using the mined sequences of programmer interactions with the Scratch programming environment and (3) reflection on the data collection and analysis effectiveness.

1.2 Related Work

We recognize that the term "Computational Thinking" has been used as an umbrella term for numerous research pursuits in promoting CS education and exploring students' computational skill practices. Kafai et al. define three main theoretical frameworks for the term—cognitive, situated, and critical—and the crucial roles they play in helping to define learning goals and evaluate pedagogical designs [11]. Our use of CT in this paper comes from a cognitive perspective to seek better understanding of students' use of computational practices. However, we believe that our proposed logging approach could be leveraged to supplement research efforts addressing CT practices from situated and critical standpoints. Logs of students' programming processes can enhance observations of students' interactions with the Scratch environment, which could then be explored alongside students' physical learning environments and cultural contexts, which, in the past, have been introduced through varying mediums such as music, textiles, and storytelling [3, 12, 10].

With an evident rise in the use of visual programming languages amongst introductory programming students, a number of studies have been conducted to assess students' CT understandings and programming practices. Several studies focus on their final code products. Maloney et al. investigated students' use of programming concepts such as abstraction by analyzing 536 Scratch projects [16]. After examining the block programs for uses of blocks such as loops, variables, and conditional statements, the authors found that students' projects lacked usage of the first three block types. Aivaloglou and Hermans performed similar analyses with a larger dataset of

over 247,000 Scratch projects, which showed conflicting results; the final codes showed increased uses of loops, conditionals, and especially variables [1].

There also exist automatic assessment tools that evaluate students' proficiency in CT using their Scratch projects. Dr. Scratch, for instance, extends Hairball, a lint-based Scratch assessment plugin [2], to detect code smells and students' uses of important programming practices based on their final codes [18]. Moreno-León et al.'s analysis of 100 Scratch projects with Dr. Scratch revealed bad programming habits such as the lack of personalized object names and code repetition. However, these analysis techniques focus only on students' final code artifacts, and thus do not consider the programming processes that led to them. This prevents assessment tools from offering students a personalized learning experience [22].

In an attempt to capture the programming processes that describe how the final code products in a novice programming environment came to be, various methods have recently been proposed. The Blackbox project has been obtaining data of students' programming actions in BlueJ, a novice-targeted Java programming environment [4]. iSnap [19] extends the Snap! visual programming environment [9] by logging students' actions during a coding task and utilizes the obtained data to automatically generate and offer personalized hints. However, to the best of the author's knowledge, there are still gaps to be filled in such efforts specific to Scratch.

Most closely related to our logging efforts is the work presented by Filvà et al., which utilizes clickstream analytics to record students' clicks in the Scratch environment [7]. Unlike our web application-based logging infrastructure, the click-based approach requires Flash, making it incompatible with mobile devices. In addition, when analyzing the obtained mouse click data, the authors take a clustering approach and identify which aspects of the interface the students interact with most frequently. In contrast, our logging and sequential pattern analysis methodology considers the order of each programming event in event sequences exhibited by students, providing more complete information about students' full programming processes.

A separate study has examined elementary students' engagement with CT by

screen recording students' programming activities and analyzing each video's content [14]. However, this approach is just as time-consuming and non-scalable as an observational study. In this thesis, we propose an automated and scalable data collection approach to collecting and analyzing data on sequences of programming events, which can help support enhancements in automated assessment and personalized learning experiences within the Scratch programming environment.

Chapter 2

METHODOLOGY AND DATA COLLECTION

2.1 Logging Programming Events

Throughout this thesis, we utilize the term 'programming event' to describe a single action taken by a user in the Scratch environment, as well as the term 'programming process' to refer to a combination of two or more programming events. The main goal of logging programming events is to gain insights into Scratch programmers' processes, enabling us to confirm or debunk researchers' expectations of novice programmers' processes, in their natural setting and without observational bias. We are interested in sequences of programmer interactions with the environment rather than focusing on the usage of a single command or transition between pairs of commands.

To achieve this goal, we needed to address several challenges: (1) determine how to best format and capture each of the programming events in the order that they occur, (2) identify additional Scratch-specific environment events that could provide insight into students' interactions with the environment beyond the actual block usage, and (3) establish an experimental environment that enables programming students to work as naturally as they would in Scratch while logging the performed events with anonymity and without interactive delay.

This section describes how we addressed each of these challenges. We first describe the anatomy and kinds of Scratch-specific events that comprise an event log. Then, we present the infrastructure we designed and implemented to gather the logs with anonymity and without interactive delay.

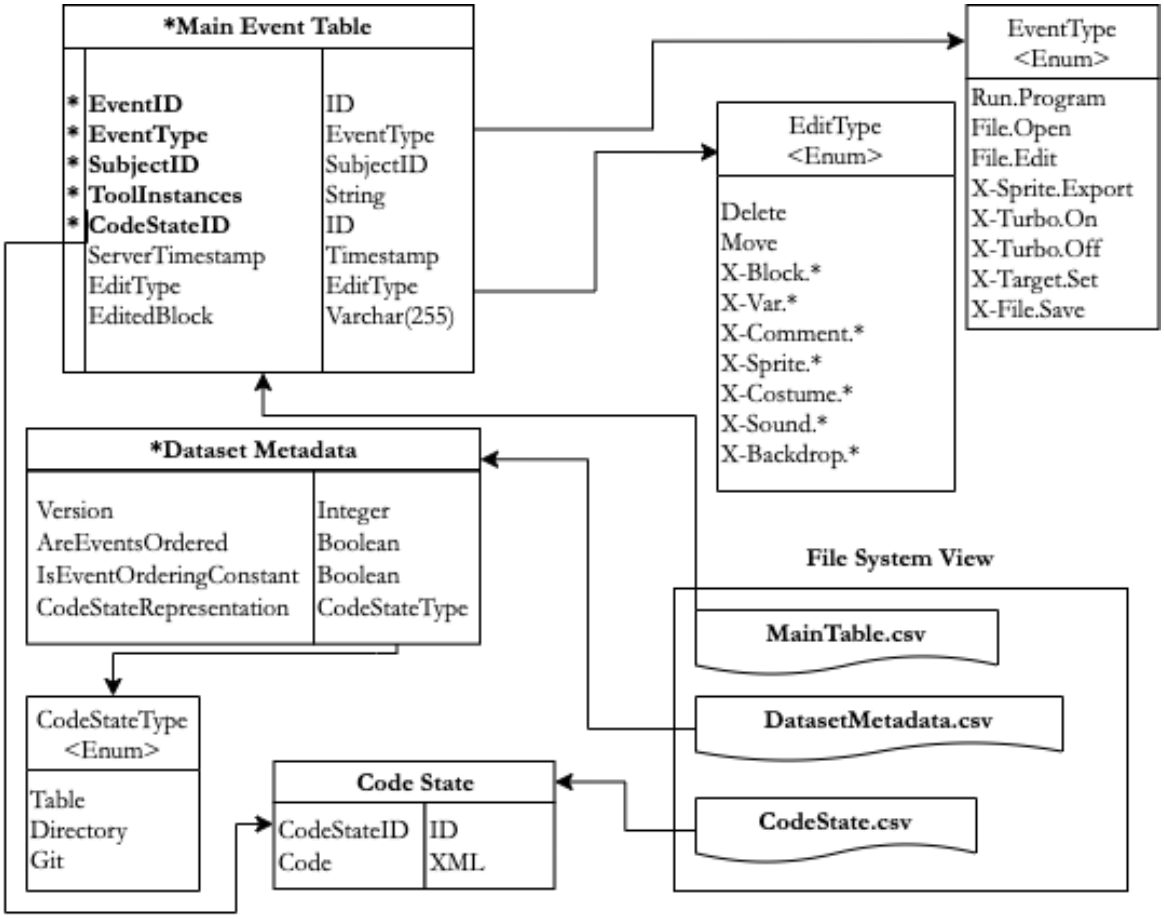


Figure 2.1: ProgSnap2-based model of programming process data in Scratch

2.1.1 The Anatomy of a Log Message

Each log message recording a student's programming event follows version 6 of ProgSnap2 specifications for datasets representing programming process data [20]. We base our logging format on the ProgSnap2 standardized format to foster sharing and collaboration among CS education researchers, which is the main goal of the ProgSnap2 creators. Figure 2.1 depicts a model of our ProgSnap2-based specification of logging messages for Scratch programming environment events.

In particular, each log message is stored as a row in the Main Event Table, which records the collection of events that occurred during the monitoring of the user's

Table 2.1: Anatomy of example log statements in the Main Event Table

EventID	EventType	SubjectID	ToolInstances	CodeStateID	ServerTimestamp	EditType	EditedBlock
2946	File.Edit	0aeW8_lW	Scratch 3.0, MySQL	1c0e1f00...	2020-02-25 11:24:34	Move	control_if
3048	File.Edit	0aeW8_lW	Scratch 3.0, MySQL	1a8b4f90...	2020-02-25 11:23:06	Move	operator_random
3049	File.Edit	0aeW8_lW	Scratch 3.0, MySQL	4f15c7e0...	2020-02-25 11:23:08	Move	sensing_askandwait

actions. For each log message, we record the unique event ID, type of event, randomly generated ID for the user being monitored, details of the tools used for the logging infrastructure, a pointer ID to the JSON representation of the current code state at the time of the event, the timestamp of when the event occurred, the type of the edit, and the edited block. Event types refer to programming environment events, while edit types refer to Scratch blocks or components. Elements that must be present in every log message in the Main Event Table are marked in bold in Figure 2.1. To illustrate, Table 2.1 depicts the log messages for a student's drag-and-drop of an `If ()` block, a random operator, and an `Ask () and Wait` block in the Scratch programming environment.

2.1.2 Scratch Edit Types

When a student performs an edit on their program, the ProgSnap2 standardized form for logging programming process data suggests that the log message specify the type of edit that was performed to provide as much detail as possible about the edit event. Since the Scratch programming environment allows interactions beyond block programming actions (e.g., sprite changes), we defined additional Scratch-specific edit types to capture events beyond block drag-and-drop actions. Figure 2.1's `EditType` shows our current complete set of edit types recorded. This includes a student's potential interaction with a sprite, costume, sound, backdrop, comment, and variable. Including such types of edits in the dataset allows us to explore if, for instance, students spend more of their time on aspects of Scratch that do not involve actual programming.

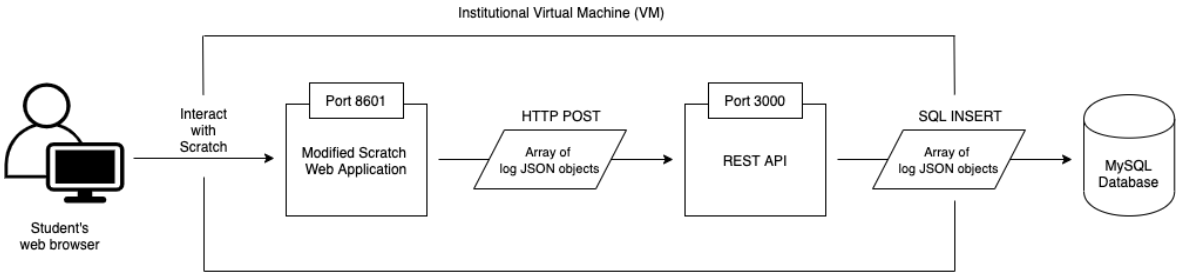


Figure 2.2: Infrastructure of our Scratch Logging Approach

2.1.2.1 Implementing the Logging Infrastructure

Figure 2 presents an overview of our infrastructure for implementing the logging process with the challenges addressed. First, to extend Scratch with logging capabilities, we modified the Scratch 3.0 programming environment, which is the latest version in use. The web application is built from several repositories, each serving a unique function. Although each repository can be installed and run on its own, the Scratch developers recommend making changes in multiple repositories when modifying the environment for development purposes. The main repositories include:

- The Graphical User Interface (GUI), which holds a set of React JavaScript-based components that make up the front-end interface,
- The Virtual Machine (VM), a library for managing the states of users' Scratch programs and keeping the GUI informed of the states,
- Scratch Blocks, a library of Scratch visual programming blocks based on Google's Blockly project, and
- The Renderer, a WebGL-based rendering engine for the environment.

For this research, we modified the GUI and VM repositories, and deployed the modified Scratch web application on a port of a university-owned machine protected by the university's firewall. This meant that the modified Scratch environment could only be loaded and displayed inside users' preferred browsers if they were connected to the university's wireless network. However, it maintained the security and anonymity of the logged data.

We also designed and deployed a separate application programming interface (API) on a different port of the machine to translate logging requests sent from the modified Scratch environment into SQL INSERT queries. These queries insert logs of a user's programming process into a MySQL-based database. The API adapts Express, a web application framework for Node.js runtime environments, and also conforms to the Representational State Transfer (REST) software architectural style, which defines a set of guiding constraints used in Web services [6].

A logging request sent from the modified Scratch environment to the REST API is in the form of an HTTP POST request with an array of JSON objects as the body. Each JSON object holds properties of a programming event that a student performed during a 90-second interval. We chose to take this approach rather than communicating with the database after each performed programming event to reduce the number of queries sent to the database storing the logs. The 90-second time interval was determined to be the most optimal after experimentation; it does not overwhelm the database, yet prevents the REST API's connection to the database from being revoked due to inactivity.

We also send a POST logging request when a student clicks on a green button to run their program, with the assumption that it is one of the last programming events that a programmer would perform towards the end of their interactions with Scratch. This is to minimize the potential loss of their event logs if they close out of their browser before the next timed POST request.

Our implementation achieves our goal of retaining the original order of events by recording the time of event occurrence under the `ServerTimestamp` column. We also maintain user anonymity in our logs by generating a non-sequential unique ID for each student accessing the modified Scratch environment. Lastly, our proposed infrastructure does not introduce any interactive delay or performance slowdowns during users' interactions with our altered Scratch application, allowing the collected programming event logs to be representative of how students would have interacted with the original website.

2.2 Mining and Analyzing Programming Processes

The main goal of our data mining is to discover interesting sequential patterns of novice Scratch programmers' processes using Scratch event logs. Our data mining approach is an adaptation of the approach of Damevski et al.'s work, which extracts usage smells and patterns in an Integrated Development Environment (IDE) among software developers [5]. Using time-ordered event logs of each student's Scratch programming events during a programming task as input, the workflow returns a list of processes commonly exhibited by the group of monitored students during that task. The output processes provide evidence on how students first approach CT coding tasks in the block programming environment, and serve as points of discussion for how students' learning experiences in Scratch could be enhanced. In addition to the mining approach, we also gain insight into how many students exhibited various anomalous block usages that are not required to successfully complete a certain coding task by extracting a set of all programming events that appeared in students' programming processes.

In this section, we first describe how we prepared the logs for mining interesting patterns, followed by how we perform sequential pattern mining. We then demonstrate the importance of identifying all programming events that were exhibited in a programming task, as well as the number of students who performed each event.

2.2.1 Preparing Data for Mining

Our mining approach only requires the Main Event Table from a ProgSnap2-based dataset. To prepare the dataset for identifying interesting programming patterns, we perform three steps.

First, it is crucial to ensure that in each student's programming process, the programming events that comprise the process are in the order of their occurrences when conducting sequential data mining. The logs in the original Main Event Table are ordered according to when they were recorded in the database for all users being monitored simultaneously during that time period. Thus, events logged for different

users are intertwined. To generate properly ordered sequences of individual students' programming events, we first separated the table by student, then ordered each student's programming process by its time order.

Second, we aggregated consecutive sets of logs that describe the identical programming event (e.g., two or more successive logs that share the same `EventType`, `EditType`, and `EditedBlock` values) into one single log message. As we are only performing mining on the Main Event Table, which does not contain the students' code states at the time of logging, repetitive logs provide limited room for interpretation. Combining such log messages helps to reduce the amount of noise in the data.

Third, we utilized the `EventType`, `EditType`, and `EditedBlock` values of each log message to create a shortened version of the log message. This step eliminates excessively detailed information that is not needed to understand each programming event. We illustrate how each student's ordered sequence of log messages is structured with an example in Figure 2.3.

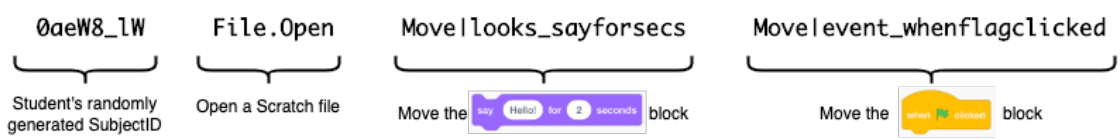


Figure 2.3: An example student's sequence of programming events

2.2.2 Sequential Pattern Mining

Once the data preparation is complete, the data to be used as input to our sequential pattern mining algorithm is in a format that holds a time-ordered sequence of each student's programming events from the Scratch programming task. Sequential pattern mining seeks to identify a set of subsequences (patterns) that occur in some percentage of the input sequences. Organizing our dataset by student (user) allows us to use the percentage as a parameter to specify a meaningful *support* (or the minimum number of times a pattern must be present in the dataset) for the mining algorithm.

The algorithm then outputs a set of patterns (or subsequences of events) that meets the predefined support requirement. For example, we used a support parameter of 10%, the same parameter utilized in Damevski et al.'s mining approach [5], to ensure that the patterns generated are substantially exhibited and are of great relevance to introductory programming students. This means that the patterns of programming events found by the algorithm should be exhibited by at least 8 of roughly 80 students to be considered commonly exhibited by the group of monitored students.

Because programming in Scratch revolves around a drag-and-drop approach, it is plausible that noise still exists in the data due to frequent clicks in the programming environment, even after preparing the original dataset for mining as described in the previous subsection. Thus, we also declared a *gap* parameter of one programming event to be used by our algorithm when establishing occurrence of potential patterns. For instance, suppose that the algorithm is attempting to find potential patterns in a student's programming sequence $x \rightarrow y \rightarrow z$. In order to consider the sub-sequence $x \rightarrow z$ as a potential pattern, the algorithm should allow for a gap of one programming event. After experimenting with other gap sizes, we determined that a gap parameter of one programming event best eliminates unrelated noise with the 10% support parameter in our specific data.

When considering candidate sequential pattern mining algorithms to use with our data, we not only took into account the two aforementioned requirements (adequately low support, gap of one programming event), but also required that the algorithm produce patterns that are *maximal*. A pattern can be deemed maximal if it does not exist within other patterns. For example, in the case that the algorithm identified the programming sequence $x \rightarrow y \rightarrow z$ as a pattern, the sub-patterns x, y, z, $x \rightarrow y$, and $y \rightarrow z$ are not included in the produced set of identified patterns. This requirement prevents the algorithm from producing an exhaustive number of sub-patterns.

Based on the requirements for our sequential pattern mining, We chose to use the MG-FSM algorithm [17] as it fulfills all the requirements, and not many of the

widely-used algorithms are capable of generating maximal patterns [8]. MG-FSM expands upon existing frequent sequence mining (FSM) algorithms by introducing gap constraints to distributed algorithms without requiring post-processing of results across partitions.

2.2.3 Manual Analysis of Mined Patterns

The primary investigators manually examined the patterns discovered by the mining step to characterize each pattern or set of similar patterns in terms of the programming process that it represents. While Damevski et al. [5] who were analyzing software developer interactions in an IDE, chose to filter out patterns that are shorter than eight events because they are less likely to offer unambiguous snapshots of a user's IDE interaction, we chose to consider patterns that are at least six events long as they provided sufficient context of what the students were directing their focus to.

To illustrate programming process patterns and their representations of programmers' potential behaviors, the sequential pattern in Figure 2.4 shows repetition of `Move` and `Delete` events involving the `Answer` block, which stores a user's answer when asked a question, similar to storing a value in a variable. Several inferences can be made from this specific pattern. For instance, students who exhibited this sequential pattern in their programming processes may have used a trial-and-error approach while experimenting with the `Answer` block. We could also deduce that the students lack a full understanding of how the block could correctly be utilized in their programs.

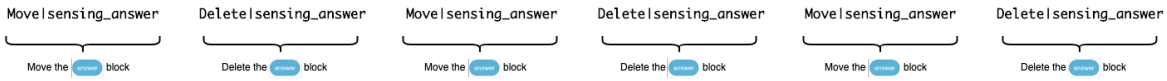


Figure 2.4: An example sequential pattern demonstrating repetitive `Move` and `Delete` events with the `Answer` block

In an iterative process of examining and characterizing mined patterns, we clustered together the patterns that we believe display the same kinds of programming behaviors. This analysis approach produced a set of programming process behaviors,

some of which deviated from what was expected from the students. We highlight some of the identified anomalous programming behaviors that were obtained in a study conducted to explore the effectiveness of our presented methodology, in the next section.

2.2.4 Deriving All Programming Events Performed

We recognize that in the block programming environment, shorter patterns have potential to reveal both common and anomalous programming behaviors, such as unexpected usage of specific blocks that may or may not remain in the final code product. For instance, a sequential pattern can be as short as two programming events, as depicted by the mined pattern in Figure 2.5. This pattern shows two drag-and-drop movements of a `Not` operator consecutively. In general, a pattern this short may not be adequate to provide context for a specific programming behavior. However, in the case where the log dataset is obtained during a particular known programming task that does not require the use of a `Not` operator, we can infer that students who demonstrated the behavior of this pattern may misunderstand logical operators at some point during their programming process.

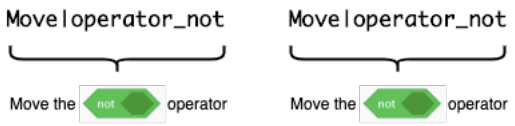


Figure 2.5: An example sequential pattern demonstrating `Move` events with the `Not` operator

Because our manual clustering procedure only considers mined patterns that are made up of 6 or more programming events, patterns that demonstrate students' use of unneeded blocks such as Figure 2.5 are overlooked. In order to identify such behavior in addition to those of the clusters, we gathered a list of all programming events that were performed at least once by students at the time of logging, along with the count of how many students exhibited each event. From this, we were able to gain a deeper,

supplemental insight into what block students interacted with that a clustered pattern may not have captured.

2.3 Applying Our Methodology to Study Students' Programming Processes

To explore the effectiveness of our approach to logging and mining of programming processes, we designed a study to identify programming novices' behaviors as they performed the same programming activity in Scratch. We then evaluated the participating students' final code products to investigate their relations with the logged processes and mined patterns. Our goal was to answer our two research questions within the context of identifying anomalous programming behaviors with respect to a given programming activity:

- **RQ1:** What kinds of anomalous programming processes are we able to expose from logging events from students during their first interactions with CT skills such as abstraction and algorithmic thinking in Scratch?
- **RQ2:** What are the costs and effectiveness of the specific logging events and pattern analysis in providing evidence of programming processes beyond what can be learned from final code products?

2.3.1 Programming Activity

We designed an in-class programming activity where students create a "Guess the Number" game. The activity consists of two subtasks, with the second subtask completed by students after they feel confident they have completed the first task. A successful program for the first subtask should choose a random number between 1 and 10, inclusive, and then repeatedly ask the player to guess the randomly generated number until they are correct. If the player's guess is too low or high, the program should inform the player accordingly, and end the game when the guess matches the random number. After students complete this first subtask, they are encouraged to attempt extending their project so it allows the player to define how many rounds of the game they would like to play and then continue generating a new random number on each round until they have played the specified number of guessing games.

The two-part activity was designed to require the use of two CT processes, abstraction and algorithmic thinking. Students' engagement with CT can be captured using programming event logs that involve constructs such as loops, variables, conditionals, and random number operators. An example of a final code product that meets all of the requirements of the activity is shown in Figure 2.7.

2.3.2 Materials

The instructions for the activity were provided orally by one of the authors and also in a written document for reference during the activity (see Appendix A). To support the students, we also created and distributed a guide that provides an overview of the Scratch programming environment's main features, along with carefully composed descriptions of a select number of constructs that we believed students would utilize the most throughout the activity (see Appendix B). Students were allowed to refer to this document as they progressed through the study. To prevent the guide from providing too much or too little background information for the activity, the document was composed and revised several times by the primary investigator and advisor of this thesis, both of whom have prior teaching experience with Scratch in introductory courses.

We also distributed a post-survey to the students at the end of the programming activity to collect their final code products, as well as information about their past computing experiences. The survey asks students about how much programming they have done prior to the study (e.g., have never programmed in any environment, tried a single of session of less than a few hours such as the Hour of Code, ...). It also asks students for a list of any Computer Science and/or Math courses that they have taken during high school or at the university. Appendix C contains a copy of the survey to illustrate its layout.

Figure 2.6: CS non-majors completing Scratch Activity

2.3.3 Procedure

With the approval of our university's Institutional Review Board (IRB), we recruited students from two undergraduate courses, each from two different departments, during the Spring 2020 semester. The Educational Technology course is designed to introduce prospective K-12 teachers technological tools that could be utilized in classroom settings, and is only offered to Education and Human Development majors. The second course, Introduction to Computer Science, is a gentle introduce to computer science primarily designed for CS non-majors. We recruited $n = 37$ students from the Ed Tech course and $n = 44$ from the Introduction to CS course to participate in our study. We chose this student population because they are students in a course intending to introduce students to block-based coding as part of the regular curriculum, and we were able to conduct our study with the participants before they reached that point in the course. This timing provided a population of novice programmers with little to no background in any kind of coding covering a variety of student majors, none of which

are computer science. Our student participant population totaled 81 participants.

For both classes, the professor of the course allowed the primary investigator to replace one of their formal lab sessions with the Scratch activity for this study. The primary investigator gave the students a 5-10 minute mini-lecture introduction to the Scratch programming environment followed by the instructions for the Scratch activity. Students were also told that the quality of their work from the activity would not have an impact on their grades. Students were then given approximately 90 minutes to perform the Scratch activity and independently create the "Guess the Number" game to the best of their abilities. Each student was provided with a computer with the modified Scratch environment ready to use, and told to work independently. When a student asked for help during the activity, they were not given direct answers and commands. Rather, they were encouraged to read the Scratch guide and try their best to connect each construct to the given instructions. Once the students had developed their own versions of the game, they were directed to submit their projects and complete the post-survey.

2.3.4 Evaluating Final Codes

After the study, the students' code submissions were assessed based on various objectives derived from the instructions. Evaluating their final code products in addition to conducting our logging and mining analysis allowed us to better investigate how the logs and mined patterns supplement the codes, as well as provide additional context to students' programming attempts. Figures 2.8 and 2.9 define the objectives used to evaluate students' submissions for the two subtasks of the programming activity respectively. Each objective describes a requirement of the programming activity.

Figure 2.7: Example solution for the programming activity

Figure 2.8: Objectives for the first subtask of the programming activity, as demonstrated by an example solution

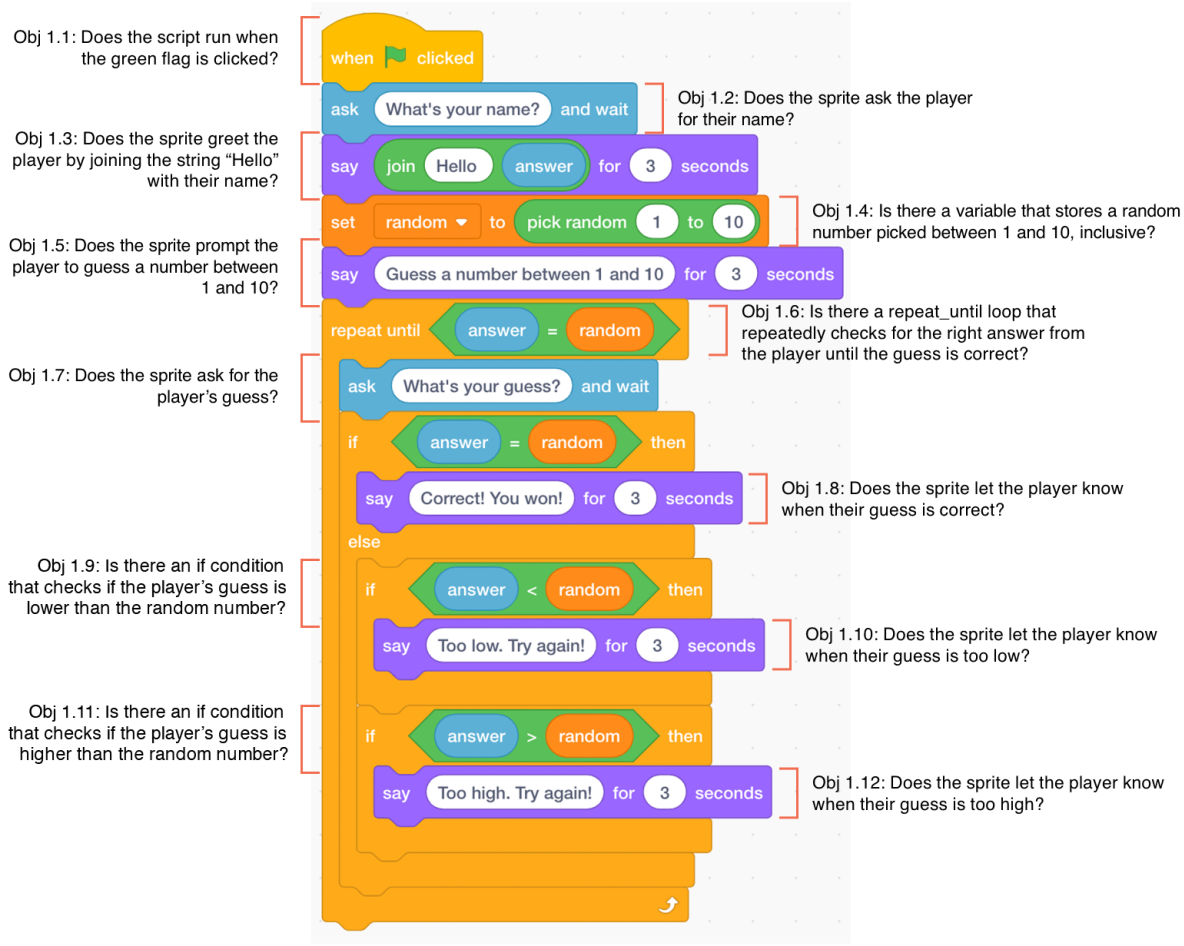
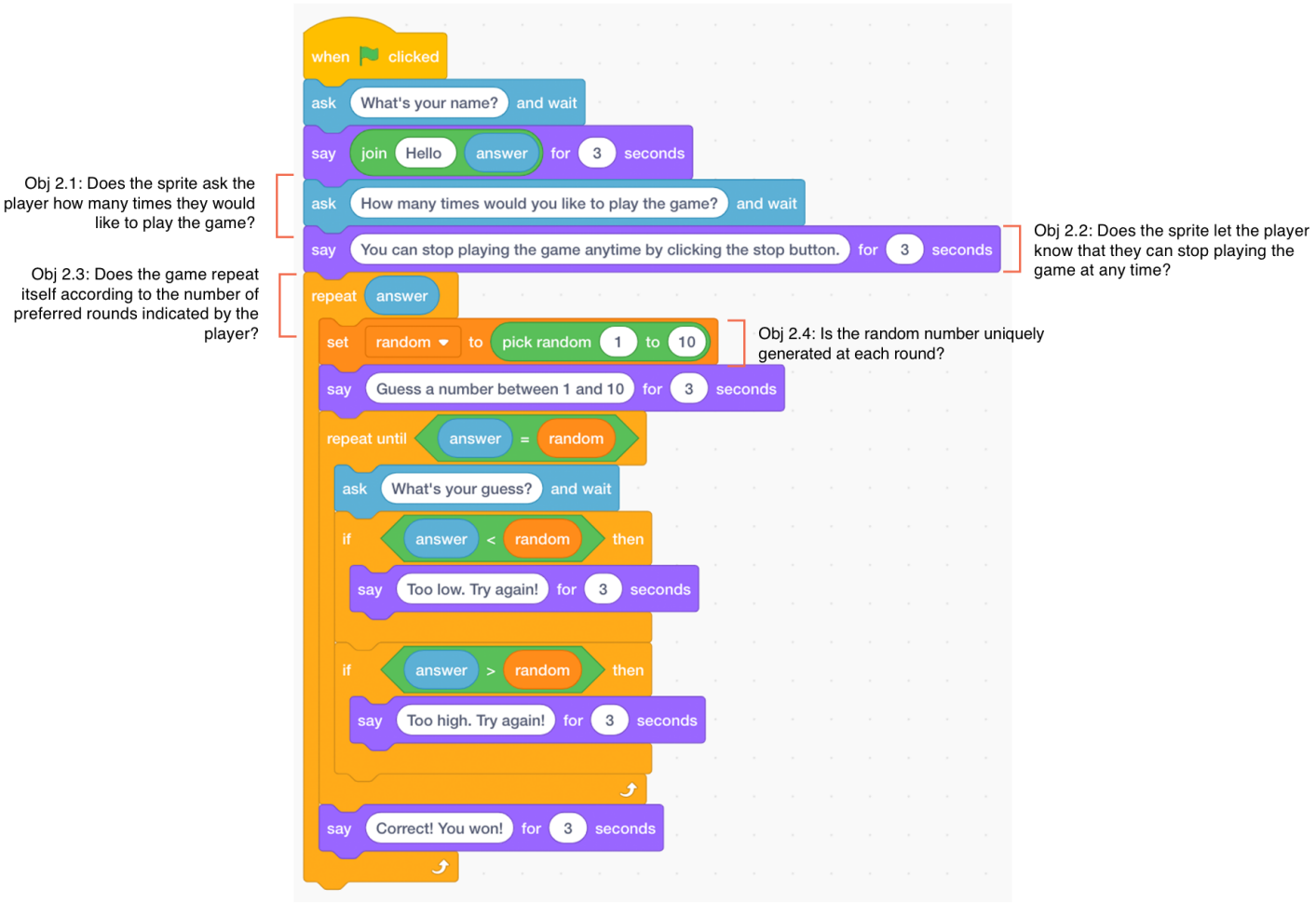


Figure 2.9: Objectives for the second subtask of the programming activity, as demonstrated by an example solution



Chapter 3

RESULTS

In this chapter, we highlight interesting logs and novice programming processes mined using our sequential pattern mining approach, towards answering our two research questions. We have made our dataset of the ProgSnap2-based programming event logs collected from the study and mined sequential patterns obtained from our sequential mining approach publicly available for other researchers to analyze[1].

3.1 Students' Programming Backgrounds

According to the students' responses to the survey, 62 of 81 students from both courses (77%) reported to have had no prior programming experience. Only about 7% of students reported to have tried a single programming session of less than a few hours, such as the Hour of Code. The remaining students indicated that they had tried creating a few projects (9%), programmed a considerable amount for a course or on their own time (5%), or programmed a fair amount in text-based languages (2%). Thus, out of the 81 students, none came into the study with a significant amount of prior programming experience in block-based languages.

3.2 Final Code Performances

We received 79 readable Scratch projects from the 81 students, which, on average, achieved 42% of the objectives from the first subtask. 54 students attempted the second subtask and achieved 43.52% of the second set of objectives on average. An outline of how many students achieved each objective of the programming activity is

[1] https://figshare.com/s/708f3707c3b2ab9e5192

Table 3.1: Students' performance on the programming activity solely based on the final code products

Objective	Description	% of Students
1.1	Script runs when the green flag is clicked	68%
1.2	Sprite ask the player for their name	91%
1.3	Sprite greets the player by joining the string "Hello" with their name	34%
1.4	A variable stores a random number picked between 1 and 10, inclusive	32%
1.5	Sprite prompt the player to guess a number between 1 and 10	48%
1.6	A `Repeat Until ()` loop checks for the right answer until the guess is correct	19%
1.7	Sprite asks for the player's guess	63%
1.8	Sprite lets the player know when their guess is correct	44%
1.9	An `If ()` condition checks if the player's guess is lower than the random number	25%
1.10	Sprite lets the player know when their guess is too low	34%
1.11	An `If ()` condition checks if the player's guess is higher than the random number	24%
1.12	Sprite let the player know when their guess is too high	32%
2.1	Sprite asks the player how many times they would like to play the game	63%
2.2	Sprite lets the player know that they can stop playing the game at any time	69%
2.3	Game repeats according to the # of preferred rounds	28%
2.4	The random number is uniquely generated at each round	15%

shown in Table 3.1. To specify an example, 19% of 79 students successfully utilized a `Repeat Until ()` loop in their game to check if the player's guess matches the randomly generated number (see Objective 1.6), and 15% of 54 students who attempted the second subtask successfully generated a unique random number at each round of the game (see Objective 2.4).

3.3 RQ1: Anomalous Programming Processes

RQ1 focuses on exploring the kinds of anomalous programming processes that we are able to expose through our logging and pattern mining approach. Table 3.2 depicts 19 clusters of log patterns that we identified by manually categorizing and clustering the generated patterns from our dataset of 81 students, following the procedure described in Sub-subsection 2.2.3. For each pattern cluster, we show the percentage of all sequential patterns obtained with our mining approach that are (1) 6 events long or longer to ensure that they sufficient context of what the students were directing their focus to, and (2) exhibit the programming process associated with the cluster. Specifically, the MG-FSM algorithm generated 2688 sequential patterns, of which 120 patterns held 6 or more programming events. For instance, the patterns comprising Cluster 3 made up 5.83% of the 120 sequential patterns that are each 6 events long or longer.

The clustered patterns in Table 3.2 are further categorized according to hypothesized representative behavior types of programming processes: (1) students' undirected approaches when working with a construct, (2) focusing on two blocks of the same type, (3) shifting focus from one block to another, (4) focusing on two blocks of different types, and (5) predominantly focusing on a block but switching focus to another block briefly.

Tables 3.3 and 3.4 highlight some of the programming events that involve blocks not required to successfully complete the activity. Events under Table 3.3 were present in the generated 2688 sequential patterns, but were not considered in the manual clustering procedure as patterns associated with the events did not meet the minimum pattern length requirement. Events in Table 3.4, on the other hand, were not present

Table 3.2: Common programming behaviors exhibited during programming activity

Programming Behavior	% of Patterns
Undirected approaches when working with a construct:	
Cluster 1 - Unsure about the `Answer` block: *move and delete*	33%
Cluster 2 - Unsure about the `Say ()` block: *move and delete*	10%
Cluster 3 - Unsure about the `Answer` block: *move block and run code to test*	6%
Cluster 4 - Unsure about the `Ask () and Wait` block: *move and delete*	2%
Cluster 5 - Unsure about the `Ask () and Wait` block: *move block and run code to test*	2%
Cluster 6 - Unsure about the `if` block: *move and delete*	<1%
Focus on two blocks of the same type:	
Cluster 7 - `Say ()` vs. `Say () for () Secs`	13%
Shift focus from one block to another:	
Cluster 8 - Shift focus from `Ask () and Wait` to `Answer` when coding	8%
Cluster 9 - Shift focus from `Answer` block to `Say ()` when coding	2%
Cluster 10 - Shift focus from `Say () for () Secs` to `Answer` when coding	2%
Cluster 11 - Shift focus from `Answer` to `Ask () and Wait` when coding	<1%
Focus on one or more blocks of various types	
Cluster 12 - Focus on the `Answer` block when coding	7%
Cluster 13 - Focus on the `Say ()` block when coding	5%
Cluster 14 - Focus on the `Answer` and `Say () for () Secs` blocks when coding	3%
Cluster 15 - Focus on the `Say () for () Secs` and `Pick Random () to ()` blocks when coding	3%
Cluster 16 - Focus on the `Answer` and `Ask () and Wait` blocks when coding	2%
Cluster 17 - Focus on the `Ask () and Wait` and `Say () for () Secs` blocks when coding	2%
Cluster 18 - Focus on the `Ask () and Wait` and `Say ()` blocks when coding	<1%
Predominantly focus on a block but switch focus to another briefly:	
Cluster 19 - Focus on the `Answer` block but briefly switch focus to the `Ask () and Wait` block	2%

in the mined patterns, but were exhibited at least once by a substantial number (more than 10%) of the students at some point during the programming activity.

From the obtained programming logs and behaviors represented by mined patterns, as well as the final code product outcomes, we identified three key anomalous behavior types:

- Undirected programming
- Misuse of blocks
- Communication-driven focus

In this subsection, we further describe the logs, mined patterns, and characteristics of the final code products that we believe exhibit each of these anomalous behaviors.

Table 3.3: Exhibited programming events that appeared in mined patterns but were not considered during clustering

	# of students who exhibited EditType	
Block Type	Move	Delete
`Wait () Secs`	44	33
`Think`	44	31
`Think for () Secs`	29	16
`Add () ()`	27	26
`Not ()`	24	21
`() and ()`	19	19
`Wait Until ()`	19	18
`When Broadcast Received`	15	15
`When Key Pressed`	12	11
`Forever`	10	9

3.3.1 Undirected Programming

Each pattern in some of the clusters that fall under this behavior type (clusters 1, 2, 4, and 6 in Table 3.2) involve several `Move` and `Delete` events of a particular construct type. When students display such patterns at any time in their programming process, we suspect that they were not fully certain about how the construct type could be best utilized.

Table 3.4: Exhibited programming events that were neither in the mined patterns nor the clusters

	# of students who exhibited EditType	
Block Type	Move	Delete
`Stop`	11	7
`Contains ()`	9	8
`Move () Steps`	9	8
`Hide Sprite`	8	7
`Show Variable ()`	8	7

Based on the block constructs involved in the anomalous behavior and the context of the assigned programming activity, we can make inferences about the students' proficiency in abstraction and algorithmic thinking. For instance, if a student exhibits the sequential pattern shown in Figure 3.1 in their programming process, it is plausible that the student was undirected about how the `If ()` block could successfully be utilized and therefore, lacked full understanding of conditional constructs.

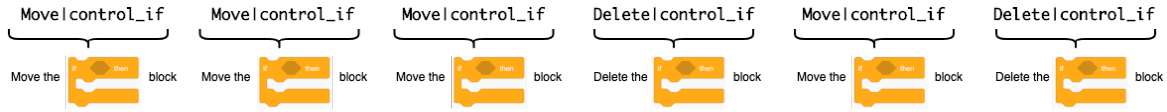


Figure 3.1: An example sequential pattern exhibiting an undirected move-and-delete approach

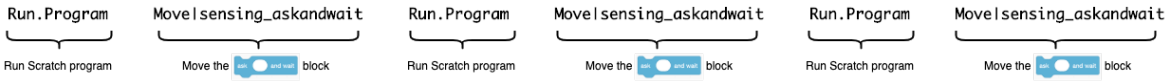


Figure 3.2: An example sequential pattern exhibiting a trial-and-error approach

Patterns in other clusters (clusters 3 and 5) were characterized to demonstrate students' trial-and-error strategy during their programming processes and involve at least 2 `Run.Program` events, which is logged when students run their Scratch programs. Figure 3.2 illustrates an example sequential pattern involving repetition of

the `Run.Program` event and the `Move` event with the `Ask and Wait` block. Such sequences suggest that students exhibiting this behavior were experimenting with the `Ask and Wait` construct and running their Scratch program after every drag-and-drop of the construct to check their codes.

3.3.2 Misuse of Blocks

Cluster 7 indicates that students at some time during their programming process used the `Say ()` command and possibly later recognized that the phrase they were trying to display to the screen did not stay on the screen long enough for the user, and some students later fixed their program to use the proper block for displaying text on the screen. This anomalous behavior was exhibited by 13% of the 120 patterns considered for clustering.

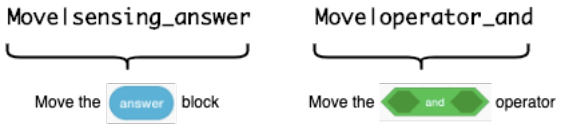


Figure 3.3: An example sequential pattern exhibiting the `() and ()` block, which is not needed in the designed programming activity

In addition, by extracting a complete list of programming events that were performed by the students during the study, some of which are highlighted in Tables 3.3 and 3.4, we were able to reveal events involving programming blocks that were neither expected nor required in our study's programming activity. For instance, the fifth block listed in Table 3.3, `() and ()`, was utilized at some time by 19 of 81 students during their programming process. Events exhibiting script edits using this block also made an appearance in mined patterns that were not manually clustered as they did not meet the minimum pattern length requirement of 6 events the clustering procedure. Figure 3.3 illustrates is an example of such pattern.

This block was not required to create the "Guess the Number" game, leading us to theorize that the students who exhibited this behavior were confusing the `() and ()`

block construct with other operators and conditionals that were required to complete the activity, such as the `() = ()` or `Join () ()` blocks. Similar hypotheses can be made about other unneeded blocks that made an appearance in students' programming processes, including the `Wait () Secs` and `Think` blocks from Table 3.3, and `Stop` and `Move () Steps` blocks from Table 3.4.

3.3.3 Communication-driven Focus

As evident by the block types that were heavily involved in the clusters of mined patterns (e.g., `Answer`, `Say ()`, `Ask () and Wait`), the students interacted the most with blocks that do not require CT skills, such as abstraction and algorithms, but rather, allow the sprite of their games to converse with the players through speech bubbles and questions. This was also demonstrated in the programming logs themselves; although they were unneeded during the activity, a substantial number of students utilized blocks that were named using speech and communication-based terms such as `Think` and `When Broadcast Received`.

This hypothesized anomalous behavior could be confirmed with the final code products. Objectives involving solely a speech bubble or a question-asking interaction were achieved by a substantial number of students (e.g., 91% for Objective 1.2; see Table 3.1) in comparison to those of CT-driven requirements (e.g., 19% for Objective 1.6). From this behavior, we can infer that students found it the easiest to understand how the communication-based blocks worked. It is also plausible that there was a greater variability in how students approached programming tasks that involved CT skills, based on the low presence of CT-involved blocks in the pattern clusters.

3.4 RQ2: Costs and Benefits Beyond Final Code Products

We first examine how the logging and pattern mining provide information about programming processes beyond what can be gained from analyzing only final code products. Then, we look at the costs associated with providing this capability.

3.4.1 Benefits Beyond Final Code Products

Indeed through our logging and pattern analysis, we were able to identify students' programming processes that were not evident in their final codes. The collected logs of programming events include details of how students approached the given programming activity and what blocks they interacted with in Scratch. This information allowed us to make inferences about whether students were detained from the main objectives of the given programming activity at any point in the study.

For instance, in the obtained logs, we noticed several programming events that represented students' drag-and-drops of motion blocks (e.g., `Move () Steps`, `Turn Left`, `Set X to ()`), which alter the positions of a sprite in the Scratch environment and were not required to create a successful "Guess the Number" game. Same could be said with regard to sensing blocks (e.g., `Key () Pressed?`, `Touching ()?`, `Color () is Touching ()?`), sound blocks (e.g., `Change Volume by ()`, `Play Sound () Until Done`), and blocks that alter a sprite's appearance (e.g., `Show`, `Switch Costume to ()`). Although none of the final projects included these blocks, we can observe that students experimented with them at a certain point during their involvement in the study.

3.4.2 Costs

The costs of a logging and sequential pattern mining technique are derived from several sources: the space needed to store the event logs, the time to monitor events which could cause noticeable lag in the interactivity during the programmers' coding session, and the time to automatically run the sequential pattern miner and to manually analyze the generated patterns. We provide insights into these costs based on our study of 81 programmers using this methodology.

For the 90 minutes of Scratch programming by 81 students, our Main Event Table containing the log message data reached a size of 34K log messages. In a MySQL-based database, the ProgSnap2-based dataset used 130 megabytes of storage. This small size shows promise for the potential for scaling to larger studies.

None of the student participants complained of any lag in the modified Scratch programming environment. We believe this is due to the small overhead in capturing the events and how our logging infrastructure does not require gigabytes of queries that approaches such as clickstream analytics may require. The MG-FSM algorithm took less than 5 seconds to generate approximately 2700 sequential patterns using the 34K logs of programming events, which we manually analyzed over the course of 1 to 2 days.

In summary, the most time consuming part of this approach is the manual analysis of the generated patterns to examine and reflect on what behaviors the patterns might be representing. The logging and pattern generation are scalable. Some of the manual analysis can be aided by automation but some requires human interpretation.

3.5 Threats to Validity

Our programming process logging and sequential pattern mining and manual analysis are susceptible to several internal and external threats. One internal threat could be in the set of events that we decided to record. We chose to follow the ProgSnap2 standard specification with adaptations for the Scratch programming environment to enable consistency with and sharing with other researchers.

We could potentially see different patterns by using a different sequential pattern mining algorithm. We chose the MG-FSM algorithm as it fulfills all of our requirements and many other sequential mining algorithms do not meet all of our requirements (See sub-subsection 2.2.2).

The manual analysis phase could lead to different conclusions as it is a human-based analysis; however, after the primary investigator performed the analysis, we mitigated the threat by both the investigator and the advisor discussing the interpretations, iterating and refining.

With regard to our data collection from students in a lab setting, although collaboration was not encouraged, some students were in close proximity to one another

and chose to discuss the activity. This may affect the individual programming behavior of those students.

With regard to external threats to validity, the results of our study of 81 novice programmers may not be generalizable to novice programmers with different backgrounds. We mitigated this threat by recruiting student from two different courses with many CS non-majors and mostly all with no experience in coding. Our study results are specific to students working in the Scratch programming environment. This may not generalize to other block-based programming environments. We utilized Scratch as it is widely used in classroom settings and many block-programming environments are similarly designed, especially considering the constructs needed to complete the programming activity we used in our study.

Chapter 4

DISCUSSION AND CONCLUSION

4.1 Discussion and Implications

With only one small programming activity that involved loops, conditionals, variables, and an interactive question-and-answer element with the user, we were able to identify clusters of patterns that could be interpreted as specific anomalous behaviors exhibited during students' attempt at the activity. The identified behaviors could also be further confirmed through a post survey designed to delve more deeply into programming processes or concepts revealed by the results of pattern analysis.

With these kinds of data, a teacher is given a window into seeing how students are approaching a given programming activity or programming in general over several activities. The patterns of anomalous behaviors could be used as points of discussions or assessment of an individual student's understanding of constructs, abstraction or algorithm development. The data for a whole class helps them gain insights into how class discussions could be directed toward concepts and programming processes with which many students in the class are struggling.

Based on what we were able to learn from our study, we believe that this logging and mining methodology has significant promise as an educational research tool. Even smaller programming activities could be designed and introduced to a larger group of student participants to collect logs of performed programming events that are carefully targeted toward learning about how novice programmers are using specific CT skills in specific contexts. We also see a potential for the methodology to be adapted to other visual programming languages. This adaptation could be leveraged to study potential similarities and/or differences among programmers learning in different programming environments.

4.2 Conclusions

In this thesis, we described how we extended the Scratch 3.0 web-based visual programming environment with logging capabilities, which record ProgSnap2-based logs of programming processes that users exhibit as they interact with Scratch. Our logging infrastructure provides detailed insight into what programming events are involved behind-the-scenes of a final code product, which fails to capture such interactions of a user. We also introduced a sequential mining approach that can derive patterns of programming processes that a substantial number of users performed, which we can manually analyze to make inferences about potential anomalous programming behaviors that the patterns may be representing. We demonstrated our approach with a study involving 81 undergraduate students enrolled in two separate courses that are introducing novice programmers to Scratch.

Based on the results from our study, our automated logging methodology successfully collected logs of students' programming events in Scratch without any noticeable delays that interfered with their interactions with the environment, showing great potential for large scalable studies in the future. The sequential pattern mining of the logs disclosed information about students' exploration of blocks that would not have been revealed from their final codes. The mined patterns also revealed the following potential programming behaviors that students may have performed during their involvement in the study: (1) programming with uncertainty in their direction or path towards a specific goal, (2) utilizing a construct that is not required in the specific programming activity, and (3) predominantly interacting with communication-based blocks that do not involve CT. We provide the dataset of programming event logs obtained from this study as contributions in addition to the approach and findings presented in this work.

4.3 Enhancing and Implementing Approach in Future Works

In future works, we plan to design smaller programming activities to assess more specific aspects of CT skills using our logging approach and explore approaches to

collecting logs of novice programmers' programming events over a longer term. Studies with such activities could be conducted with both introductory programming students and CS educators learning about the Scratch environment. Exploring the programming processes of both the students and the instructors can allow us to understand how novice programmers first interact with CT from multiple perspectives.

With a larger dataset, we will also be investigating automated clustering procedures to enhance our analysis methodology. This could be utilized to train a machine learning model, and potentially lead to developments of Intelligent Tutoring Systems (ITSs) and other forms of automatically personalized learning tools specific to the Scratch environment.

In addition, our approach may be of great help to qualitative efforts that study the physical learning environments themselves, such as classrooms and after-school programming clubs in K-12 schools, and how identity, social, and cultural contexts play a role when delving into the CS field for the first time. Logs of participants' programming processes can supplement practices such as interviews and observations. To allow the methodology to be utilized beyond a university setting while maintaining the security of the logged data, however, the current infrastructure would need to be modified and improved upon.

Results obtained from our exploratory study serves as a crucial discussion point as to how novice programmers' learning experiences, as well as their learning and retention of CT skills, can be enhanced. Efforts toward doing so could be in the form of introducing tools or curricula to supplement novice programming specific to Scratch, improving the design and structures of block-based programming environments, and developing an alternative visual programming language for introductory CS students that may not necessarily revolve around the idea of blocks.

REFERENCES

[1] Efthimia Aivaloglou and Felienne Hermans. How Kids Code and How We Know: An Exploratory Study on the Scratch Repository. In *Proceedings of the 2016 ACM Conference on International Computing Education Research*, ICER '16, pages 53–61, New York, NY, USA, 2016. ACM.

[2] Bryce Boe, Charlotte Hill, Michelle Len, Greg Dreschler, Phillip Conrad, and Diana Franklin. Hairball: Lint-inspired static analysis of scratch projects. In *SIGCSE 2013 - Proceedings of the 44th ACM Technical Symposium on Computer Science Education*, pages 215–220, 2013.

[3] Karen Brennan, C Balch, and M Chung. Creative computing 3.0. *Cambridge, MA: Harvard University*, 2019.

[4] Neil C.C. Brown, Amjad Altadmri, Sue Sentance, and Michael Kölling. Blackbox, five years on: An evaluation of a large-scale programming data collection project. In *ICER 2018 - Proceedings of the 2018 ACM Conference on International Computing Education Research*, pages 196–204. ACM, 2018.

[5] Kostadin Damevski, David C. Shepherd, Johannes Schneider, and Lori Pollock. Mining Sequences of Developer Interactions in Visual Studio for Usage Smells. *IEEE Transactions on Software Engineering*, 43(4):359–371, 2017.

[6] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation, University of California, Irvine, 2000.

[7] Daniel Amo Filvà, Marc Alier Forment, Francisco José García-Peñalvo, David Fonseca Escudero, and María José Casañ. Clickstream for learning analytics to assess students' behavior with Scratch. *Future Generation Computer Systems*, 93:673–686, 2019.

[8] Philippe Fournier-Viger, Antonio Gomariz, Ted Gueniche, Azadeh Soltani, Cheng Wei Wu, and Vincent S. Tseng. SPMF: A java open-source pattern mining library. *Journal of Machine Learning Research*, 15:3389–3393, 2015.

[9] Dan Garcia, Brian Harvey, and Tiffany Barnes. The beauty and joy of computing. *ACM Inroads*, 6(4):71–79, Dec 2015.

[10] Fatemeh Jamshidi and Daniela Marghitu. Using music to foster engagement in introductory computing courses. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, SIGCSE '19, page 1278, New York, NY, USA, 2019. Association for Computing Machinery.

[11] Yasmin Kafai, Chris Proctor, and Debora Lui. From theory bias to theory dialogue: Embracing cognitive, situated, and critical framings of computational thinking in K-12 Cs education. In *ICER 2019 - Proceedings of the 2019 ACM Conference on International Computing Education Research*, pages 101–109, New York, New York, USA, jul 2019. Association for Computing Machinery, Inc.

[12] Yasmin B Kafai, Deborah A Fields, Debora A Lui, Justice T Walker, Mia S Shaw, Gayithri Jayathirtha, Tomoko M Nakajima, Joanna Goode, and Michael T Giang. Stitching the loop with electronic textiles: Promoting equity in high school students' competencies and perceptions of computer science. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, pages 1176–1182, 2019.

[13] Sze Yee Lye and Joyce Hwee Ling Koh. Review on Teaching and Learning of Computational Thinking Through Programming. *Comput. Hum. Behav.*, 41(C):51–61, Dec 2014.

[14] Sze Yee Lye and Joyce Hwee Ling Koh. Case studies of elementary children's engagement in computational thinking through scratch programming. In *Computational Thinking in the STEM Disciplines: Foundations and Research Highlights*, pages 227–251. Springer International Publishing, Jan 2018.

[15] David J Malan and Henry H Leitner. Scratch for Budding Computer Scientists. *SIGCSE Bull.*, 39(1):223–227, Mar 2007.

[16] John H Maloney, Kylie Peppler, Yasmin Kafai, Mitchel Resnick, and Natalie Rusk. Programming by Choice: Urban Youth Learning Programming with Scratch. *SIGCSE Bull.*, 40(1):367–371, Mar 2008.

[17] Iris Miliaraki, Klaus Berberich, Rainer Gemulla, and Spyros Zoupanos. Mind the gap: Large-scale frequent sequence mining. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 797–808, 2013.

[18] Jesús Moreno-León, Gregorio Robles, and Marcos Román-González. Dr. Scratch: Automatic Analysis of Scratch Projects to Assess and Foster Computational Thinking. *RED-Revista de Educación a Distancia*, 2015.

[19] Thomas W. Price, Yihuan Dong, and Dragan Lipovac. isnap: Towards intelligent tutoring in novice programming environments. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '17, page 483–488, New York, NY, USA, 2017. Association for Computing Machinery.

[20] Thomas W Price, David Hovemeyer, Kelly Rivers, Austin Cory Bart, Andrew Petersen, Brett A Becker, and Jason Lefever. ProgSnap2: A Flexible Format for Programming Process Data. In *Proceedings of the Educational Data Mining in Computer Science Workshop in the Companion Proceedings of the International Conference on Learning Analytics and Knowledge (LAK 2019)*, pages 1–7, 2019.

[21] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. Scratch: Programming for All. *Commun. ACM*, 52(11):60–67, Nov 2009.

[22] Gregorio Robles, Jean Hauck, Marcos Román-González, Roberto Nombela Alonso, Christiane von Wangenheim, and Jesús Moreno-León. On Tools that Support the Development of Computational Thinking Skills: Some Thoughts and Future Vision. In *Proceedings of International Conference on Computational Thinking Education (CTE 2018)*, 2018.

[23] Jeannette M Wing. Computational Thinking. *Commun. ACM*, 49(3):33–35, Mar 2006.

Appendix A

STUDENTS' COPY OF THE INSTRUCTIONS FOR THE PROGRAMMING ACTIVITY

Scratch Programming Activity [Students' Copy]

Your challenge

In this programming task, you will be creating a 'guessing game' with Scratch. At the beginning of the game, your program should choose any random number from 1 to 10. Then, your program should repeatedly ask the player to guess the number until they are correct, informing them if the guessed number is too high or too low. The game ends when the player guesses the random number correctly.

Instructions — Part I

Click on the following link to access the website that will be used for this activity: **http://128.4.31.241:8601/**

Once you have made your way to the website we will be using for this programming activity, click on the 'Session Code' button on the top menu, which will give you your very own session code. Copy and paste your session code into the Google Form, then return to the website to begin the Scratch programming task.

A. Greet the player

When your guessing game starts, the Scratch Cat sprite should first ask the player for their name by saying "What is your name?". When the player provides a name, instruct the Scratch Cat to say "Hello [insert name here]" for 3 seconds to greet the player with the given name.

B. Pick a random number

After greeting your player, your program should randomly choose any number between 1 and 10. This is the number that the player will attempt to guess.

C. Tell the player to guess the number

Program the Scratch Cat so that it prompts the player to guess the randomly generated number by saying 'Guess a number between 1 and 10' for 3 seconds. The Cat should then ask 'What's your guess?' and allow the user to answer the question. If the player's guess is correct, instruct the Cat to say "Correct! You won!" for 3 seconds. Otherwise, the Cat should repeatedly ask the player to guess again and inform the player that they have guessed too high or too low by saying "Too low. Try again!" and "Too high. Try again!" accordingly for 3 seconds.

D. Submit your project

Please save your Scratch project to your computer and attach it to the Google Form.

Using the ‘File’ menu found at the top of the window, click on ‘Save to your computer’ to save the project to your computer.

Before moving on from Part I to Part II

Does the project you created so far satisfy all of the instructions above?

Yes:

Answer accordingly on the Google Form and move onto Part II.

No:

Answer accordingly and download the example Scratch project provided on Google Form.
To load the example Scratch project into the Scratch programming environment, click on the ‘File’ menu found at the top of your window. Then click on ‘Load from your computer’ to load the project into the environment.

Instructions — Part II

Extend your project so that it accomplishes the following:

A. Ask the player how many times they would like to play the guessing game

In the first section of part I, your program controlled the Scratch Cat sprite to greet the player with the given name, then generated a random number from 1 to 10. For part II, modify your program so that after the player is greeted with their name, the Scratch Cat asks, “How many times would you like to play the game?”. The player should provide a number in response to the sprite’s question.

B. Tell the player that they can stop the game anytime

Before generating a random number, program your Scratch cat to say “You can stop playing the game anytime by clicking the stop button” for 3 seconds.

C. Pick a random number and repeat the guessing game ___ times

Using the number that the player has given, your program should repeat the guessing game that many times. The random number should also be regenerated each round.

D. Submit your project

Please save your Scratch project to your computer and attach it to the Google Form.
Using the ‘File’ menu found at the top of the window, click on ‘Save to your computer’ to save the project to your computer.

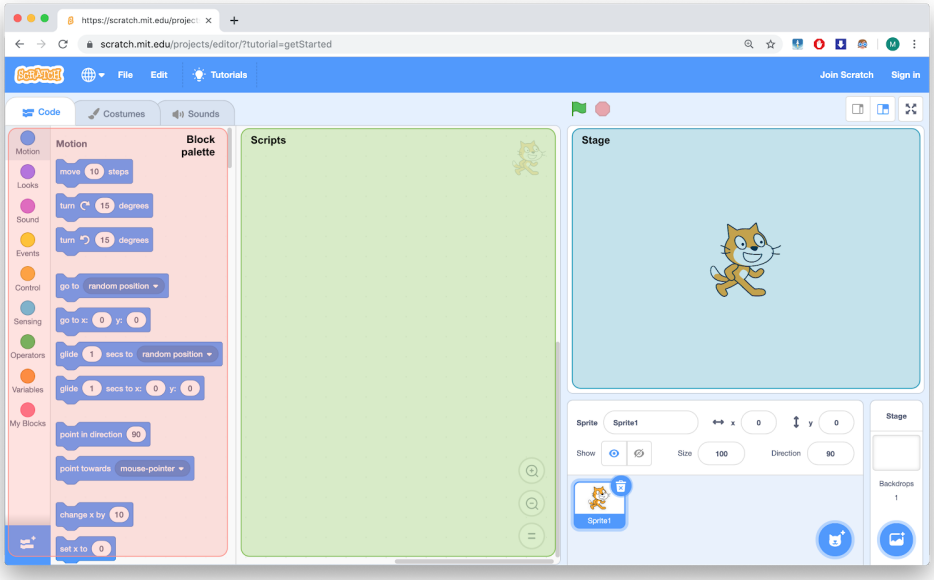
Appendix B

AN OVERVIEW GUIDE TO THE SCRATCH PROGRAMMING ENVIRONMENT

Getting Started with Scratch

Overview of the Scratch programming environment

For the programming challenge, you'll be working with three main areas in the Scratch programming environment: the block palette, the scripts area, and the stage.



The **block palette** contains every block that can be used in a Scratch project. Click and drag blocks into the scripts area to run them in your project. Blocks can also be stacked together to create a *script*, which is a sequence of instructions that your project will follow.

The **scripts area** stores your project's scripts. This is where you will be writing your script with the blocks in the block palette.

The **stage** is a visual representation of your project. What is shown on the stage is what a user will see when they interact with your work. When Scratch creates a new project, its stage will come with the Scratch Cat sprite by default (see above screenshot).You will be working with this Scratch Cat sprite for this guessing game.

Blocks

Looks

: Give your sprite a speech bubble and make it say something by using the **Say ()** block.

Events

: **When the Green Flag is clicked**, all block codes that are attached to this Event block will run.

Controls

: Blocks held inside this **Repeat ()** block will repeat a given number of times, before allowing the script to continue past that section of the script.

: Blocks held inside this **Repeat Until ()** block will repeat until the condition described in the hexagon becomes true.

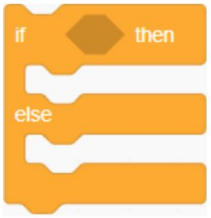
E.g. 

: A sprite will move 10 steps repeatedly until its horizontal (x) position equals 100.

: Blocks held inside this **Forever** block will repeat an infinite number of times (the loop never ends). You can end the **Forever** block using the **Stop ()** block.

: The **Stop ()** block is used to stop the script it is attached to, all scripts in the project, or other scripts in a sprite.

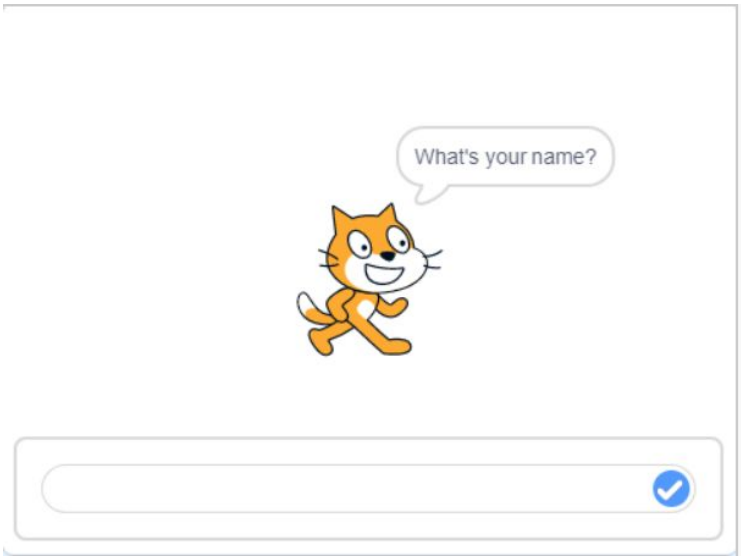
: The **If () Then** block checks the condition described in the hexagon.. If the condition is true, the blocks held inside it will run. If the condition is false, the blocks held inside will be ignored, unlike the **if () Then, Else** block.



: The **If () Then, Else** block checks the condition described in the hexagon. If the condition is true, blocks held under ***If () then*** will run. However, if the condition is false, blocks held under ***Else*** will run.

Sensing

: With the **Ask () and Wait** block, you can ask your players questions and receive answers. Specify the question that you would like to ask into this block. When this block runs, it will display the question and an empty text box, in which your players can type in their answers.



: Once the players submit their answers, they are stored in the **Answer** block.

Operators


: The **() + ()** block adds the two values.

: The **() - ()** subtracts the second value from the first value.

: The **() * ()** multiplies the two values.

: The **() / ()** block divides the second value from the first value.

: The **Pick Random () to ()** block picks a random number "ranging from the first number to the second, including both endpoints."

: The **() is greater than ()** *boolean* block checks if the first value is greater than the second value. If the first value is greater, the block returns ***true***. If the second value is greater, the block returns ***false***.

: The **() is less than ()** *boolean* block checks if the first value is less than the second value. If the first value is greater, the block returns ***false***. If the second value is greater, the block returns ***true***.

: The **() = ()** *boolean* block checks if the first and second values are equivalent to one another. If the values are equal, the block returns ***true***. Else, the block returns ***false***.

: The **() and ()** *boolean* block "joins two boolean blocks so they both have to be true to return ***true***." If both boolean blocks are *true*, the block returns ***true***. If only one or none of the boolean blocks are true, the block returns ***false***.

: The **() or ()** *boolean* block "joins two boolean blocks so any one of them can be true to return ***true***." If at least one of the two boolean blocks are *true*, the block returns ***true***. If none of the two boolean blocks are true, the block returns ***false***.

: The **Not ()** *boolean* block returns ***true*** if the boolean inside it is false; if the boolean inside it is true, it returns ***false***.

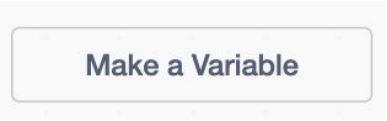
: The **Join ()()** block "links" the two specified values together. For example, if "hello" and "world" were put in the block, the **Join()()** block would

return “helloworld”. To form a coherent sentence, insert a space in one of the two values (e.g. “hello “ and “world”, “hello” and “ world”).

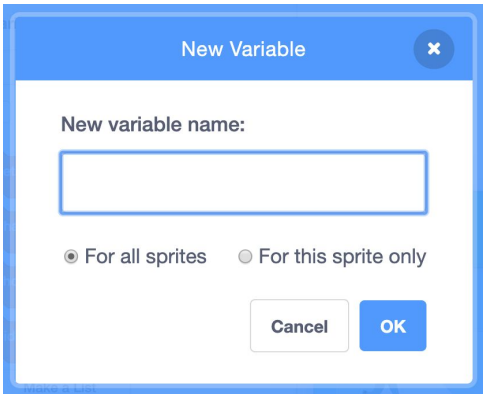
Variables

: Sometimes you may need to store information that would be useful later in your Scratch project. In programming, a **variable** is like a box that can store different values like numbers and words.

To create a variable, look for the ‘Make a Variable’ button in the block palette.



A ‘New Variable’ window such as the following should appear in your Scratch programming environment:



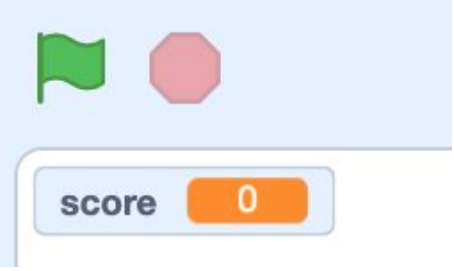
Give your variable a name and specify whether you would like this variable to be used for all sprites or this sprite only. Click ‘OK’ when complete. You should now be able to store values in your new variable.

: The **Set () to ()** block sets the specified variable to store the given value. The value can be in the form of numbers or words. For example, you can store number 0 in a variable named ‘score.’

: The **Change () by ()** block changes the value stored in the specified variable by the given number. If the variable was storing word(s) before this block is called, the **Change () by ()** block changes the value stored in the variable to the given number. In this screenshot, the value stored in the ‘score’ variable changes by 1.

: The **Show Variable ()** block shows what value the specified variable is holding in the stage of your Scratch programming environment.

E.g. When this block is called in your script and it specifies a variable named ‘score’ storing the number 0, the stage displays the value of ‘score’ on the top left corner as such:



: The **Hide Variable ()** block hides the value of the specified variable from the stage of your Scratch programming environment.

Appendix C

SURVEY UTILIZED IN EXPLORATORY STUDY

Welcome to the Scratch programming activity!

Note that this activity is designed for everyone, regardless of one's programming background.

* Required

Terms and Agreements: Consent Form

You are being asked to participate in a research study. This form tells you about the study including its purpose, what you will be asked to do if you decide to participate, and any risks and benefits of being in the study. Please read the information below and ask the research team questions about anything we have not made clear before you decide whether to participate. Your participation is voluntary and you can refuse to participate or withdraw at any time without penalty or loss of benefits to which you are otherwise entitled. If you decide to participate, you will be asked to sign this form and a copy will be given to you to keep for your reference.

WHAT IS THE PURPOSE OF THIS STUDY?
The purpose of this study is to observe Scratch users' use of computational thinking skills, such as abstraction and algorithmic thinking. We need beginning programming students such as you to help us gain a detailed understanding of users' common programming behaviors in the Scratch block-coding environment.
You are being asked to take part in this study because Scratch is designed to address beginning programmers. We do not require, or want you to have significant programming experience. A log of your actions in the Scratch programming environment will be recorded, but no information regarding your identity will be collected. This anonymous log, along with many others, will be analyzed to learn about the actions that beginning programmers take when block-coding in Scratch. The aggregate results from many users will be reported in research papers. We plan to have about 30-60 participants in this study.

WHAT WILL YOU BE ASKED TO DO?
You will first be given an overview of the Scratch programming environment.
Then, you will be given an introductory programming activity to complete in Scratch. You will be asked to follow the procedures below:
1. Complete and submit a Google form provided by us, where you will be asked to provide information such as your level of proficiency in programming.
2. Complete the 'Getting Started with Scratch' guide.
3. Create, to the best of your abilities, a Scratch program that follows the instructions given to you.

Participation in the study will take approximately 20-45 minutes.

WHAT ARE THE POSSIBLE RISKS AND DISCOMFORTS?
There are no known risks to you if you participate in this experiment. If you refuse to participate, you will not be penalized.

WHAT ARE THE POTENTIAL BENEFITS?
Participants will gain block-programming experience and knowledge that they may not have had prior to partaking in the study, and could make use of in future programming activities. You will be helping researchers in computer science education to better understand how beginners go about programming, which can lead to new ways of teaching beginning programmers.

HOW WILL CONFIDENTIALITY BE MAINTAINED?
We will make every effort to keep all research records that identify you confidential to the extent permitted by law. In the event of any publication or presentation resulting from the research, no personally identifiable information will be shared outside the lab.
Your research records may be viewed by the University of Delaware Institutional Review Board, but the confidentiality of your records will be protected to the extent permitted by law.

WILL THERE BE ANY COSTS RELATED TO THE RESEARCH?
There are no costs involved in participation.

DO YOU HAVE TO TAKE PART IN THIS STUDY?

Participation in this study is voluntary. It is your decision as to whether you want to participate. If you refuse to participate, you will not be penalized. If at any point during the experiment, you change your mind and wish to withdraw, you may do so. If you request to have the data collected from you destroyed, we will do so.
As a student, if you decide not to take part in this research, your choice will have no effect on your academic status or your grade in a class.

WHO SHOULD YOU CALL IF YOU HAVE QUESTIONS OR CONCERNS?
If you have any questions about this study, please contact the Principal Investigator, Lori Pollock at 302-831-1953 or email pollock@udel.edu.
If you have any questions or concerns about your rights as a research participant, you may contact the University of Delaware Institutional Review Board at 302-831-2137.

Your signature below indicates that you are voluntarily agreeing to take part in this research study. You have been informed about the study's purpose, procedures, possible risks and benefits. You have been given the opportunity to ask questions about the research and those questions have been answered. You will be given a copy of this consent form to keep.

1. **Signature of participant** *

2. **Date signed** *

   Example: December 15, 2012

Take a quick look at what you can do in Scratch

Make your way through the 'Getting Started with Scratch' guide. The guide provides an overview of the Scratch programming environment and the various block codes that you may be using throughout the study. Click on the following link to access the 'Getting Started with Scratch' guide:
https://docs.google.com/document/d/1De4s5Fdlbp69YuDwYkkb7PJyDBQQQDbi16FQEOVU4sc/edit?usp=sharing

3. *
   Check all that apply.

   - [ ] Click this checkbox after you have read through the 'Getting Started with Scratch' guide.

Programming Activity: Part I

Follow the instructions for part I of the programming activity in the following document:
https://docs.google.com/document/d/10Mt1VusUMPTPvMkZZgNd07TECQPY0gI8m1YTv9sPeHA/edit?usp=sharing

4. **Copy and paste your session code here:** *

5. **Upload your Scratch project here after completing part I of the programming activity.** *
   Files submitted:

Before moving on

6. **Does the project you created so far satisfy all of the instructions above?** *
   Mark only one oval.

   - Yes *Skip to question 8.*
   - No *Skip to question 7.*

Answered 'No' to 'Does the project you created so far satisfy all of the instructions above?' from Part I?'

Part II requires you to have a successfully working code from Part I.
Download the example solution Scratch project by clicking on the following link:
https://drive.google.com/file/d/1dIkTU76rMPaSh55ixZEk3oxUM4SkcWRq/view?usp=sharing

Once you have downloaded the example project, load the .sb3 file into your Scratch programming environment.

7. *
   Check all that apply.

   - [ ] Click this checkbox after you have successfully loaded the example [solution] project into your Scratch programming environment.

Skip to question 8.

Programming Activity: Part II

Follow the instructions for part II of the programming activity in the following document:
https://docs.google.com/document/d/10Mt1VusUMPTPvMkZZgNd07TECQPY0gI8m1YTv9sPeHA/edit?usp=sharing

8. **Upload your Scratch project here after completing part II of the programming activity.** *
   Files submitted:

Some questions about you

9. **How much programming experience do you have?** *
   Mark only one oval.

   - Have never programmed in any environment
   - Tried a single session of less than a few hours (e.g. Hour of Code)
   - Created a few projects
   - Have programmed a fair amount in block-based languages
   - Have programmed a fair amount in text-based languages
   - Have programmed a considerable amount (e.g. as part of a course, during my own free time)

10. **What Computer Science courses have you taken in high school and UD?** *

_____

_____

_____

_____

_____

11. **What math courses have you taken in high school and UD?** *