The Common Coder's Scratch Programming Idioms and Their Impact on Project Remixing

Xingyu Long
Software Innovations Lab
Dept. of Computer Science
Virginia Tech
Virginia, USA
xingyulong@cs.vt.edu

Peeratham Techapalokul*
College of Integrated Science and Technology
Rajamangala University of Technology Lanna
Chiang Mai, Thailand
peeratham@rmutl.ac.th

Eli Tilevich
Software Innovations Lab
Dept. of Computer Science
Virginia Tech
Virginia, USA
tilevich@cs.vt.edu

Abstract

As Scratch has become one of the most popular educational programming languages, understanding its common programming idioms can benefit both computing educators and learners. This understanding can fine-tune the curricular development to help learners master the fundamentals of writing idiomatic code in their programming pursuits. Unfortunately, the research community's understanding of what constitutes idiomatic Scratch code has been limited. To help bridge this knowledge gap, we systematically identified idioms as based on canonical source code, presented in widely available educational materials. We implemented a tool that automatically detects these idioms to assess their prevalence within a large dataset of over 70K Scratch projects in different experience backgrounds and project categories. Since communal learning and the practice of remixing are one of the cornerstones of the Scratch programming community, we studied the relationship between common programming idioms and remixes. Having analyzed the original projects and their remixes, we observed that different idioms may associate with dissimilar types of code changes. Code changes in remixes are desirable, as they require a meaningful programming effort that spurs the learning process. The ability to substantially change a project in its remixes hinges on the project's code being easy to understand and modify. Our findings suggest that the presence of certain common idioms can indeed positively impact the degree of code changes in remixes. Our findings can help form a foundation of what comprises common Scratch programming idioms, thus benefiting both introductory computing education and Scratch programming tools.

*This research was conducted when the author was a postdoctoral fellow at the Software Innovations Lab, Dept. of Computer Science, Virginia Tech

SPLASH-E '21, October 20, 2021, Chicago, IL, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-9089-7/21/10…$15.00
https://doi.org/10.1145/3484272.3484959

CCS Concepts: **• Software and its engineering → Patterns**; **Visual languages**; **• Social and professional topics → Computing education**.

Keywords: Programming idioms, Scratch, Block-based programming, Novice programmers, Project remixing

ACM Reference Format:
Xingyu Long, Peeratham Techapalokul, and Eli Tilevich. 2021. The Common Coder's Scratch Programming Idioms and Their Impact on Project Remixing. In *Proceedings of the 2021 ACM SIGPLAN International SPLASH-E Symposium (SPLASH-E '21), October 20, 2021, Chicago, IL, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3484272.3484959

1 Introduction

A *programming idiom* describes a recurring syntactic code pattern that implements a specific functionality. Because idiomatic code is conducive to good programming styles, programmers find programs that contain primarily idiomatic code easier to understand [26]. Similar to the effectiveness of design patterns in computing education pedagogy [17], recognizing programming idioms can be seen as an important part of introductory computing education. For more advanced learners, the awareness of common programming idioms can also help in transferring programming knowledge across languages.

Scratch has been a highly successful programming language and a learning community for novice and end-user programmers [24]. Currently, the Scratch community has over 68 million registered users with 72 million shared projects [6]. The block-based visual nature of Scratch enables introductory learners to quickly become familiar with basic programming knowledge and coding skills, so the language has become a mainstay of introductory computing classes in all levels [32]. Despite the abundance of Scratch tutorials and other learning materials, the prevalence and usage of common Scratch programming idioms remain unexplored.

In this paper, we bridge this knowledge gap by systematically studying this issue.

Software engineering researchers and practitioners alike commonly use and widely apply the term *programming idioms*. Programming idioms usually emerge in a bottom-up fashion from common use. However, extracting highly recurring patterns from a large number of Scratch projects might not be a suitable strategy to follow in order to satisfactorily identify common programming idioms in this language. Novice programmers often write recurring pieces of code that reflect their unawareness of better alternatives, or even misconceptions and bad habits. Therefore, we chose to extract recurring patterns from canonical sources, programming textbooks authored by seasoned Scratch programmers, so we could use their crystallized knowledge and experience as our source of common programming idioms.

Understanding common programming idioms is important to be able to effectively use and learn a language. Some studies have indicated that certain aspects of the Scratch's open-ended learning model might not be as effective for all learners as previously thought, and why models that employ more guided learning can be beneficial [22]. Some studies suggest that students might not deeply engage with those programming concepts that educators expected them to master [7, 18]. The highly interactive programming model of Scratch empowers students to learn through trial and error by tinkering with source code, in order to associate different code fragments with the program output. In the process, students may end up producing non-idiomatic code patterns and embracing them as part of their coding practices. In fact, writing non-idiomatic code repeatedly might have led learners to poor quality code as observed in prior studies, such as code smells [30] and bugs [20]. Steering students in the direction of using idiomatic coding patterns may help them in effectively mastering the fundamentals of programming, so they can quickly progress toward more advanced concepts.

In fact, the concept of patterns, which is embodied in programming idioms, in introductory programming is not new. A growing body of research focuses on anti-patterns (e.g., code smells, poor coding practices)—that is, patterns that should be avoided or refactored. For example, Frädrich et al. found that software bugs afflict a large portion of Scratch projects [20]. However, the opposite of anti-patterns, as with programming idioms in Scratch, has been relatively unexplored.

The objective of this work is to better understand Scratch programming idioms by answering the following questions:

- **RQ1:** What are the common programming idioms in top 5 Scratch programming books?
  By *common*, we mean *comprehensive*. That is, we identify programming idioms as they pertain to both (1) general programming tasks and (2) most popular Scratch-specific programming domains (e.g., media computing, games, animations, etc.).
- **RQ2:** How common are those idioms in a large, diverse dataset of Scratch projects?
  Our goal is to understand how common these idioms are across the different categories of Scratch projects. We created a tool for detecting the identified idioms and applied it to a large number of Scratch projects to determine the prevalence of these idioms.
- **RQ3:** How does the code change within the detected common idioms when projects are remixed?
  Because certain idioms are conducive to program comprehension and modification, understanding how these idioms impact projects remixing can guide the design of educational strategies. We investigate how code changes within two types of detected idioms related to control flow (i.e., Nested IF Else and Forever IF/IF Else) between the original projects and their remixes. We selected these two idioms due to them being related to control-flow, with high prevalence across different project categories. Their control-flow structures isolate the differences introduced by remixes, so we could reliably observe how these idioms manifest themselves in remixing practices.

We identified a total of 11 common idioms that belong to 2 main categories: 1) Scratch-specific (i.e., Change And Clone, Change And Wait, Repeat Change, Sensor Wait Until No Sensor, Switch Backdrop And Broadcast, and Broadcast And Stop) and 2) general programming idioms (i.e., Iterate List, Delete All Items By Value, Forever IF/IF Else, Nested IF Else, and Repeat Ask). By analyzing Scratch projects for the presence of the aforementioned idioms, we discovered the prevalence of these idioms. Additionally, by investigating how code changes between original projects and their remixes within the detected Nested IF Else and Forever IF/IF Else idioms, we discovered that the majority of these changes are code deletions. It suggests that programmers often remove code blocks in their remixing projects to achieve their goal and they are more likely to maintain simpler projects.

This paper makes the following contributions:

1. A catalog of 11 common Scratch programming idioms, including their definitions and usage scenarios
2. An automated tool for automatically detecting programming idioms; we extended Litterbox[20], a state-of-the-art framework for analyzing Scratch programs
3. A large-scale study that assesses the prevalence of the studied programming idioms based on a diverse dataset of over 70K projects
4. A case study of code changes within two common control-flow idioms detected in the original projects and their remixes

The remainder of the paper is organized as follows: Section 2 presents background information on programming idioms and Scratch. Section 3 provides a summary of prior studies on idioms and patterns in Scratch. Section 4 explains our approach used to answer the research questions. Section 5 presents the findings of our study. Section 6 discusses the significance and implications of our findings as well as the threats to validity of the study in Section 7. Finally, Section 8 presents future work direction and concluding remarks.

2 Background

In this section, we describe the background information about programming idioms and Scratch, required to understand our technical contributions.

Programming Idioms: A programming idiom refers to a commonly recurring pattern of code in a given language. Since programming idioms reflect prevailing programming styles, they serve as a bottom-up type of programming knowledge, acquired via hands-on programming activities. Structurally, a programming idiom can comprise a single or multiple code fragments. In general, a programming idiom represents a simple functionality that the underlying programming language does not directly provide as built-in libraries. The same programming idiom is often present in different programming languages, but manifests in different forms based on the specific language's programming style (e.g., 'iterating over a range:' 'for (int i = 0; i < 10; i++)' in Java and 'for i in range(0, 10)' in Python).

A number of research works study programming idioms. Early work dating back into 1970's suggests that programming idioms serve as a useful programming concept that can be applied to teaching programming to students [27]. Other studies aim at understanding how idiomatic code is used and how it impacts software. Alexandru et al.[8] built a catalog of 'Python idioms' having discovered how they greatly improve code quality. Smit et al. [28] analyzed a large volume of open-source projects and discovered that several coding-conventions positively impact maintainability.

Scratch: Released in 2007, Scratch has since become highly popular among novice programmers in both formal and informal educational settings [24]. Its intuitive visual programming interface allows programmers to compose a program by snapping jigsaw-like blocks together. Unlike in general text-based programming languages, such as Java and Python, learners in Scratch can quickly learn core programming concepts without the need to master the language's syntax. Scratch makes learning programming engaging by empowering learners to create media-rich projects that include animations, games, tutorials, art, music, and stories [1]. Scratch fosters a learning community, in which programmers learn from each other by sharing and remixing projects. This practice can be seen as a form of code reuse, similar to forking in the context of professional software development.

A Scratch program comprises programmable objects (the Stage and its children objects known as "sprites"). The Stage and sprite objects are controlled by a set of scripts, which is a sequence of blocks. Scratch has made several design choices to keep their language features simple. For example, a Scratch procedure has no return value and cannot be shared among different sprite objects. Although minimal in its design, the Scratch language introduces novice programmers to several fundamental programming concepts, including sequences, flow controls, variables, procedural abstractions, synchronizations, and Boolean conditions.

Another advantage of Scratch is a wide availability of learning materials, including books, freely available programming tutorials, and community-curated programming tips. We used these materials as canonical examples of what constitutes "good" Scratch programming style and our source of idioms.

3 Related Work

In this section, we review the most closely related prior research efforts.

Our study of programming idioms in Scratch is among many prior research efforts that aim at better understanding and supporting programmers in this domain. Related works in this area apply program analysis to Scratch, such as the studies of code smells [30] and software bugs [20]. A common approach to automatically analyze coding patterns is to leverage static program analysis, which operates on the abstract syntax tree (AST) representation of a program. Like programming idioms, bug patterns and code smells exhibit unique structural characteristics that can be exploited by automatic detection approaches. Although designed for different purposes, some of the prior works provide open-source reusable program analysis infrastructures to build upon. In particular, we extended the core functionality of LitterBox [20], a state-of-the-art Scratch program analysis framework, to implement our analyzer of programming idioms.

Several prior works have also focused on programming idioms in Scratch. In particular, Amanullah and Bell conducted a series of studies of programming idioms referred to in their work as "elementary programming patterns". Their early work proposed two main categories of elementary programming idioms— loop[15] and selection[16] ('If' related) patterns —as a way to help students avoid programmings problems (e.g., code smells, bug patterns) and assess the prevalence of these patterns in a large dataset of Scratch projects [10]. These findings show that elementary patterns are uncommon among projects created by inexperienced programmers [11]. Their later work suggests that remixing can

[1]https://en.scratch-wiki.info/wiki/Project_Types

help programmers learn elementary patterns. Specifically, when instances of elementary pattern were detected in an original project, the same pattern instances were also found, often in greater number, in its project remixes [12]. Their most recent work applies elementary programming patterns to measure how comprehensive a set of Scratch teaching materials and introductory Python books are in covering their patterns [13].

Our work differs from the prior research conducted by Amanullah and Bell in a number of ways. Firstly, our goal is to comprehensively document common Scratch programming idioms, as based on a representative corpus of canonical educational materials. Specifically, we draw our idioms from popular Scratch educational materials, considering not only general programming but also Scratch-specific idioms. Secondly, our large-scale study involves dissimilar sets of projects as subjects to better understand existing patterns of using common idioms. Specifically, we applied our analysis to projects that were authored by programmers with dissimilar levels of programming experience and coming from different backgrounds (i.e., based on trending projects, classroom studios, top Scratch programmers, etc.) and categories (i.e., games, animations, etc.). Finally, we investigated the types of changes in the idioms' body between projects and their remixes. Our case study can thus shed light on how project remixes use these idioms, with the goal of promoting the Scratch communal learning.

4 Methods

In this section, we describe the approaches we used to answer our research questions.

4.1 Identifying programming idioms

To identify programming idioms, the first and the second authors worked collaboratively by following a two-stage process:

Stage 1: Gathering Sources Canonical sources of programming idioms were compiled from online Scratch programming lessons and books. A collaborative document was kept, with entries for each found programming idiom. Each entry included screenshots of the idiom's instances, their location (e.g., URL, page number), and comments describing its usage context. This shared document allowed both authors to organize and keep track of any recurring instances or new idiom candidates discovered so far.

When identifying idioms or structural patterns in code, one has no choice but to rely on their own experience and understanding of the studied language's constructs, their semantics as well as how they are used in practice to achieve specific functionalities. Hence, this process is necessarily subjective. To minimize subjectivity in our identification process, we carefully examined the repeated patterns in code written by experienced Scratch programmers. That code would be expected to be idiomatic. We also used the common characteristics of idioms as widely discussed in the literature as a guideline in classifying each repeated pattern as a potential idiom. Specifically, we looked for small, reusable, structured coding patterns written to achieve a particular purpose. This strategy led us to identifying potential candidates whose frequencies we used to come up with our final list of idioms in Stage 2.

Although not all of the examined Scratch resources contained idioms, we were still able to compile our final catalog of 11 idioms. Some of these resources turned out to be not a good source of idioms, such as online Scratch programming lessons that often present open-ended instructions. These exploratory and experimental learning materials reflect a Constructionist design, the driving learning philosophy of Scratch. For instance, Google's "CS First" offers several project-based lessons that focus on a step-by-step guide with minimal boilerplate code as a starting point[2].

Other types of resources proved to be reliable sources of idioms that made our final list. As is usually the case, programming books in any language can be typically relied on to contain coding idioms. These books cover similar materials, thus reflecting shared communal knowledge. Specifically, Scratch programming books often explain fundamental programming topics (sequences, variables, control flow, etc.) by presenting the source code of example projects in major project categories (games, animations, stories, etc.). These books represent the main sources that we used when identifying the programming idioms for our study.

Stage 2: Identifying Idioms Each recorded idiom was evaluated as a candidate to be included into our list of idioms. Both authors strived to achieve a mutual understanding of each idiom and its usage context. By resolving disagreements, the authors reached a consensus on whether to include candidate idioms in the catalog. They considered not only how frequently an idiom recurs across the sources, but also if other popular programming languages (Java, Python, etc.) have corresponding idioms, as reflected in a crowd-sourced, online catalog of idioms curated for major text-based languages [5]. This stage also provided an opportunity to tentatively name and initially define the idioms' generic form, as a starting point of a further iterative refinement process.

The included idioms were divided into two categories: *general programming* idioms and *Scratch-specific*. We classified any idioms that appear in other programming languages as *general programming* idioms. *Scratch-specific* idioms come from usages that perform specific media-computing functionalities, such as animation, art, games, music, stories. Overall, we discovered that when it comes to Scratch books, many of them are published by the O'Reilly Media. Our sources are the five Scratch textbooks which represent the minimal set of sources that contain the studied idioms.

[2]https://csfirst.withgoogle.com/c/cs-first/en/curriculum.html

Next, we briefly provide a summary of the five aforementioned books. These books contain a variety of content, both general and domain-specific. They provide representative examples for identifying recurring Scratch programming idioms associated with programming concepts (e.g., conditional statement, for-loop statement, variables, operators, and etc) and project categories (e.g., animations, art, games, music, stories, tutorials).

For example, some of this educational content introduces the fundamentals of Scratch programming (e.g., "Hello Scratch!: Learn to program by making arcade games [19]" and "Make Your Own Scratch Games! [14]") by explaining how to make computer games. These textbooks provide well-written coding examples, but may have different target audiences. For example, major game design concepts and implementation strategies would be more appropriate for advanced Scratch programmers, while the basics of writing games in Scratch would be of primary interest for beginners.

As another example, "Scratch by Example: Programming for All Ages [31]" introduces the basics of Scratch as well as illustrates how several advanced programming concepts should be used, including the list data structure, customized blocks, and webcam interaction. "Scratch 3 Programming Playground [29]" provides a large number of idiomatic coding samples for different common programming tasks by creating several game-related projects. Finally, "Learn to Program with Scratch [25]" presents the material centered around different block types (e.g., a comprehensive overview and usage examples of each block).

4.2 Studying the prevalence of Scratch programming idioms

To assess the prevalence of programming idioms, we collected a large representative sampling of Scratch projects from the "trending" category. We obtained our list of trending projects via the project fetching API that allows specifying different query parameters (e.g., `https://api.scratch.mit.edu/explore/projects?limit=16&offset=0&language=en&mode=trending&q=animation`). We chose to focus on trending projects, as they tend to be more mature in comparison to the projects in the "recent" category. Trending projects are also likely to be non-trivial, as indicated by their high visibility among community members, thus making it possible to exclude projects that contain only graphical media without any code.

To better understand how idioms are used, we collected additional project samples of different groups of programming backgrounds to compare with the average trending projects. Specifically, we focused on two specific experience backgrounds 1) Scratch classroom studios, comprising projects created by students in formal classroom settings and 2) Top community-favorite programmers, comprising projects created by highly experienced Scratch programmers in the community. After removed the empty projects (i.e., zero block found in projects), we collected a total of 70K projects during March, 2021 (29,771 projects for top Scratchers, 43,340 projects for trending, 730 projects for studio).

We also collected projects from high school and college classes, which we referred to as *studio* dataset. *Studio* is a project collection in Scratch, a feature commonly used in a classroom setting to collect and organize projects created by students. This feature is also used by programmers in the Scratch community to curate a collection of projects sharing a similar genre or theme. In our study, we carefully chose a subset of studios that appear to associate with a classroom setting. In particular, we collected 25 different classes which include computer science courses from high school and college level and 730 projects in total.

For RQ3, we used a small subset of Scratch projects that have been remixed (forked) multiple times and contained two common control-flow idioms: Forever IF/IF Else and Nested IF Else. Because the analysis required comparing the original projects with their remixes, we collected a random sample of twenty remixes for each studied original project. Not all of the projects met the criteria of having 20 or more remixes. In summary, the RQ3 dataset comprises a total of 342 projects (59 original projects and their 283 remixes).

Overall, the collected project samples represent all six considered categories (e.g., animations, games, tutorials, art, music, and stories)[3]. Scratch allows programmers to tag shared projects with a category name.

4.3 Detecting Programming Idioms

We extended LitterBox [20] to implement a set of analysis routines, each detecting a specific programming idiom. Source code for our implementation was published at Github (`https://github.com/xingyu-long/LitterBox`)

An idiom detection routine operates on the abstract syntax tree (AST) of a parsed Scratch project. It traverses the AST, collects each visited node's information to determine whether the idiom is present based on its definition, adding the detections to a final report.

For idioms with sequential structures (e.g., Change And Clone, Change And Wait, and Broadcast And Stop), we apply our detection tool to all statement sequences contained in different AST nodes with a nested structure (e.g., 'forever', 'repeat', 'if', etc). For each idiom with complex structures, we develop a customized detection strategy. For example, Nested IF Else includes multiple 'If-else' code fragments according to our definition. Our tool detected this idiom by traversing the nested structure of AST nodes to keep track of the number of 'If-Else' nodes encountered and reported whether this structure exists. To make sure that we exhaustively cover all edge cases, we developed a suite of comprehensive test cases for each variety of our idiom detectors.

[3]https://en.scratch-wiki.info/wiki/Project_Tags

LitterBox's modular design made it easy to extend to create customized traversal strategies for accessing the target program elements of the analyzed idioms. Having clearly defined idioms, experienced developers should be able to translate the definitions to working code that collects all the necessary program information needed to detect idioms.

4.4 Computing programming idioms metrics

To assess the prevalence of an idiom, we calculated the percentage of the projects in the dataset that contains at least one instance of the idiom. We performed this calculation for the projects in different project categories (e.g., games, animations, storytelling, etc.) and three sample datasets representing different programming experiences/backgrounds (i.e., (1) trending (general), (2) top Scratchers, and (3) studio projects).

We explored how code changes within the body of the Forever IF/IF Else and Nested IF Else idioms across the original projects and their remixes. We detected these idioms in the original projects and then collected their block IDs and identified the corresponding code fragments in their remixes. We converted the code fragments to textual representations to make them amenable to differencing as plain text. Then, we calculated the type of changes in terms of deletion, insertion, and update, represented as percentages of each operation.

5 Findings

In this section, we present the findings that answer our research questions.

5.1 Common Programming Idioms in Scratch (RQ1)

Identified by following the procedure outlined above, the following idioms are cataloged into general programming and Scratch-specific groups.

General programming idioms:

1. Iterate List: Similarly as in other programming languages, a *list* is a useful data structure for storing multiple pieces of information. Scratch provides a set of basic command blocks for reading a list value at a given index as well as manipulating the stored values (e.g., adding and deleting item by its value from the list). This idiom is used to iterate the list values and perform actions on each accessed value [1]. Fig 1) shows an example of this idiom.
2. Delete All Items By Value: Scratch supports storing in a list values of string and number types only. Additionally, all operations on a list are based on either the item indices or the item values. This specific list-based idiom removes all list items that matches a specified item value [2]. Fig. 2 shows the generic form of this idiom.

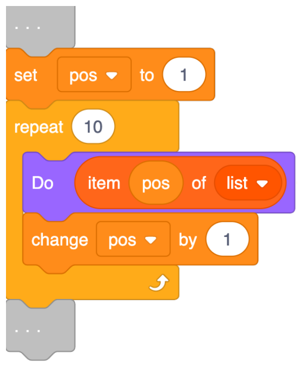

Figure 1. Iterate List

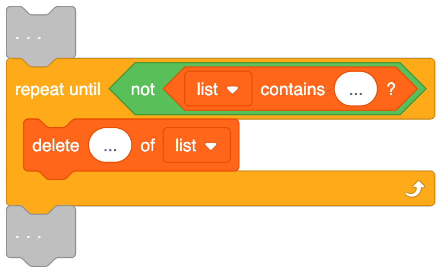

Figure 2. Delete All Items By Value

3. Forever IF/IF Else: This idiom provides a way to continuously monitor a given Boolean condition [25]. For *if-then* inside the body of *forever* block, it perform no action when the condition is false and perform specified actions whenever the condition is true. The need for this usage scenario is so common that historically Scratch introduced it as one of its control blocks but later removed it in Scratch 2.0 [3]. A less common variation of this idiom, when *if-then-else* block is used in place of *if-then* block, provides a way to also perform specific actions continuously when the monitored condition is not met.
4. Nested IF Else: Selecting one among multiple code fragments to execute based on their conditions is a common programming idiom in any language [25]. Some programming languages allows multiple *else if* to be inserted between *then* and *else* parts to specify additional conditions. In Scratch, this multi-branch control structure is achieved by repeatedly nesting the *if-then-else* block within the *else* part of the previous *if-then-else* block. Fig. 3 shows an example of this idiom.
5. Repeat Ask: Scratch provides the `ask () and wait` block to prompt for text input from the user [25]. The most recent text input is then stored in the "answer" block. This idiom performs input validation by continuously asking for user input until a valid text input is submitted [25]. Fig.4 shows an example of this idiom.

Scratch-specific Programming Idioms:

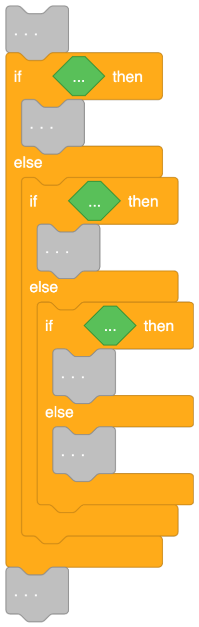

Figure 3. Nested IF Else

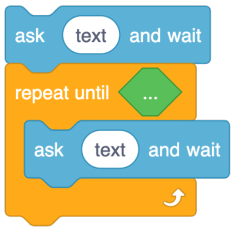

Figure 4. Repeat Ask

6. Change And Clone: Creating clones is analogous to instantiating objects, a Scratch clone inherits attribute values (e.g., local variable, position, visibility, etc.) from a sprite parent, which can be viewed as the clone's prototype. This idiom sets sprite attributes that become the initial attribute values of a soon-to-be created clone. The idiom contains one or more side-effect command blocks (e.g., `change [variable v] by (1)`) that precede the `create clone of [myself]. Fig.` 5 shows an example of this idiom.

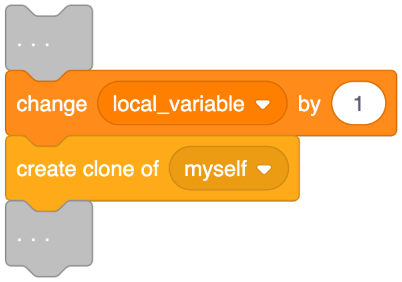

Figure 5. Change And Clone

7. Change And Wait: This idiom introduces delay in between command blocks that have immediate effects on the program output (e.g., setting an object's position). Executing these blocks in sequence would result in instant changes that are imperceptible to the human eye. To create an illusion of smooth changes over a period of time (e.g., a moving object), a wait block (e.g., `wait 0.01 sec`) is used to insert a small delay between each included block (or a sequence of blocks that needs to be executed without delays) [19]. Fig. 6 shows an example of this idiom.

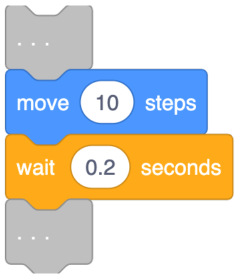

Figure 6. Change And Wait

8. Repeat Change: Scratch projects are often made up of several low-level, elementary animated elements that can be combined to create increasingly complex animations, often used by storytelling and game projects. This idiom is used in many elementary animations (e.g., fading a sprite's transparency, growing a sprite's size) by repeating a sequence of side-effect causing blocks (e.g., position, size, graphic effects) a specified number of times. Fig. 7 shows an example of this idiom.

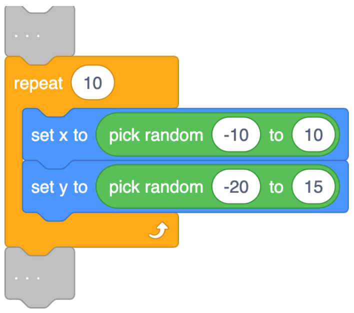

Figure 7. Repeat Change

9. Sensor Wait Until No Sensor: To handle user input events, an infinite loop (i.e., `forever` block) is often used alongside *sensor* blocks to monitor specific user input events. However, using an `if` block to check whether a user input event (e.g., mouse clicks, key presses) is present can trigger event-based code more than once at a time, resulting in an unwanted program behavior in some scenarios. This idiom ensures that a stream of events of the same type will not re-trigger event-based code by using a `wait until [sensor block]` block to wait for the absence of the event first. Fig. 8 shows an example of this idiom.
10. Switch Backdrop And Broadcast: The Stage serves not only as a parent object for all sprites in a project, but also as a visual container that renders the background (called a backdrop in Scratch). This idiom sets a backdrop as the visual context, and then coordinates sprites accordingly via a broadcast block.
11. Broadcast And Stop: This idiom diverts program flow from one script to other scripts when a certain condition is met. This idiom coordinates various parts of a program in response to a specific program state. Fig. 9 shows a common use case of this idiom. A script handles an exceptional case

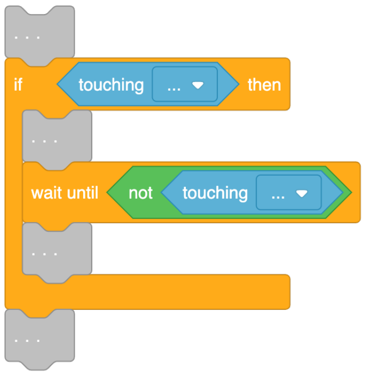

Figure 8. Sensor Wait Until No Sensor

in which an error message is broadcast and the execution of that script is stopped.

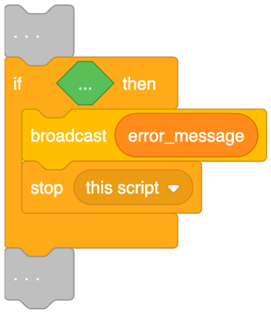

Figure 9. Broadcast And Stop

5.2 Prevalence of programming idioms (RQ2)

Table 1 provides the statistical summary of the *trending* dataset. We calculate the mean, standard deviation, and five-number summary of the basic Scratch program elements (blocks, procedures, scripts, and sprites) in 43,340 project samples. The size of the project samples in terms of number of blocks varies widely, as indicated by a high standard deviation of *numBlocks*. On average, a project in our dataset contains 374 blocks, 29 scripts, and 7 sprites.

Table 2 presents the prevalence of each programming idiom in three datasets: *trending*, *top Scratchers*, and *studio projects*.

Overall, each programming idiom is similarly prevalent across the three datasets. The top three most prevalent idioms are Change And Wait, Repeat Change, and Forever IF/IF Else, respectively. Each of these three idioms are present in over one third (33%) of project samples. Notably, the top two idioms are often present in more than half of the project samples.

Moderately prevalent idioms are Change And Clone and Nested IF Else. These idioms are present from 10% to 21% of the project samples except in the studio dataset where they are less prevalent (5.75%). Broadcast-related idioms (i.e., Switch Backdrop And Broadcast and Broadcast And Stop) are less prevalent, with most of them present in less than 5% of the projects, although Switch Backdrop And Broadcast is present at a relatively higher percentage (9.18%) in the studio dataset. Finally, list-related idioms (i.e., Delete All Items By Value, Iterate List) and Repeat Ask are uncommon, detected in less than 0.3% of the sample projects in all three datasets.

From the studio dataset, Sensor Wait Until No Sensor, Delete All Items By Value, and Iterate List show no prevalence (0%). A possible explanation is that introductory Scratch programming courses often adopt an exploratory style for coding assignments. Also, many courses might introduce students to Scratch programming early in the curriculum and then transition to text-based languages, thus reducing the opportunities for students to apply more advanced Scratch idioms.

To better understand the distribution of programming idioms, we further investigated how prevalent the identified idioms are in the trending dataset across different project categories. Table 3 presents the results. Change And Wait and Repeat Change, commonly used idioms for creating animation, are similarly prevalent across different project categories, especially in animations, games, and stories. These idioms were detected in a range of 41% to 77% of the projects. The prevalence of Forever IF/IF Else varies noticeably across project categories. Specifically, less than 28% of the animation, art, and storytelling projects contain this idiom, while 74.44% of games and 53.18% tutorials projects contain this idiom at least once. Especially, those two idioms are less than 10% in the art category, and they show 3.15% for Nested IF Else and 7.71% for Change And Clone.

Broadcast-related idioms (Switch Backdrop And Broadcast and Broadcast And Stop) appear in less than 10% of projects in each category. We found that only Broadcast And Stop in game projects is 9.43%, which is higher than Broadcast-related idioms in other project categories. The idioms related to the list data structure (i.e., Iterate List) are uncommon, found only in stories, appearing in only 0.01% of all projects. Other uncommon idioms include Sensor Wait Until No Sensor, and Repeat Ask whose prevalence is less than 4% across project categories, used in less than 1% of all projects.

5.3 Idioms and code changes through remixing (RQ3)

We explored how code changes within the body of two common control-flow idioms. We considered three types of operations: block insertion, block deletion, and value update.

Fig. 10 shows a partial example of the analyzed code changes. The original and its remixed instance of the Forever IF/IF Else idiom appear on the left and right sides, respectively. This example shows how code changed in two

Statistic	N	Mean	St. Dev	Min	Pctl(25)	Median	Pctl(75)	Max
numBlock	43,340	374.62	966.50	1	25	99	347	28031
numProcedure	43,340	2.77	9.94	0	0	0	1	285
numScript	43,340	29.13	65.68	0	4	10	29	2314
numSprite	43,340	6.77	10.01	0	2	4	8	332

Table 1. Basic summary statistics of trending dataset

Programming idiom	Trending	Top Scratchers	Studio
Change And Wait	60.23%	58.43%	64.79%
Repeat Change	58.31%	63.96%	42.19%
Forever IF/IF Else	39.79%	40.86%	52.88%
Change And Clone	17.54%	20.48%	5.75%
Nested IF Else	10.69%	12.68%	5.75%
Broadcast And Stop	2.86%	3.82%	4.25%
Switch Backdrop And Broadcast	2.52%	3.01%	9.18%
Sensor Wait Until No Sensor	1.14%	1.72%	0
Delete All Items By Value	0.10%	0.14%	0
Repeat Ask	0.05%	0.08%	0.27%
Iterate List	< 0.01%	0.03%	0

Table 2. Prevalence of programming idioms in trending, top Scratchers and studio datasets

Programming idiom	% in Animations	% in Art	% in Games	% in Music	% in Stories	% in Tutorials
Change And Wait	76.62%	41.45%	65.50%	48.46%	73.26%	51.76%
Repeat Change	70.38%	42.51%	70.67%	49.08%	68.60%	46.46%
Forever IF/IF Else	26.92%	23.55%	74.44%	32.18%	27.11%	53.18%
Change And Clone	19.70%	7.71%	30.51%	13.41%	21.92%	11.27%
Nested IF Else	9.20%	3.15%	22.87%	6.21%	14.11%	7.87%
Broadcast And Stop	1.78%	0.80%	9.43%	2.27%	0.96%	2.30%
Switch Backdrop And Broadcast	3.66%	1.06%	2.78%	1.7%	3.31%	2.24%
Sensor Wait Until No Sensor	0.82%	0.39%	3.18%	0.74%	0.92%	0.85%
Delete All Items By Value	0.03%	0.05%	0.43%	0.06%	0.01%	0.05%
Repeat Ask	0	0	0.21%	0.01%	0.06%	0.01%
Iterate List	0	0	0	0	0.01%	0

Table 3. Prevalence of programming idioms in each project category subset within the trending dataset

places. The first change is the insertion of two more blocks within the "if-then" block. The other change is the modified Boolean expression of the existing "if" block, with extra operator blocks that include several sensing-related blocks. Our analysis captures these changes as 3 insertions.

We analyzed a total of 367 instances of Nested IF Else and 739 instances of Forever IF/IF Else. The two programming idioms have dissimilar characteristics of code changes across the original projects and their remixes. Tables 4 and 5 summarize the change operations that occurred within the body of the analyzed idiom instances. The results on both of these tables show that the majority of code changes are deletions with ~82% for Nested IF Else and ~54% for Forever IF/IF Else. Although the percentages of deletions in both idioms are similarly high, the percentages of insertions are markedly different. Block insertions account for almost 27% of all code changes in Forever IF/IF Else as compared to about 5% in show no Nested IF Else.

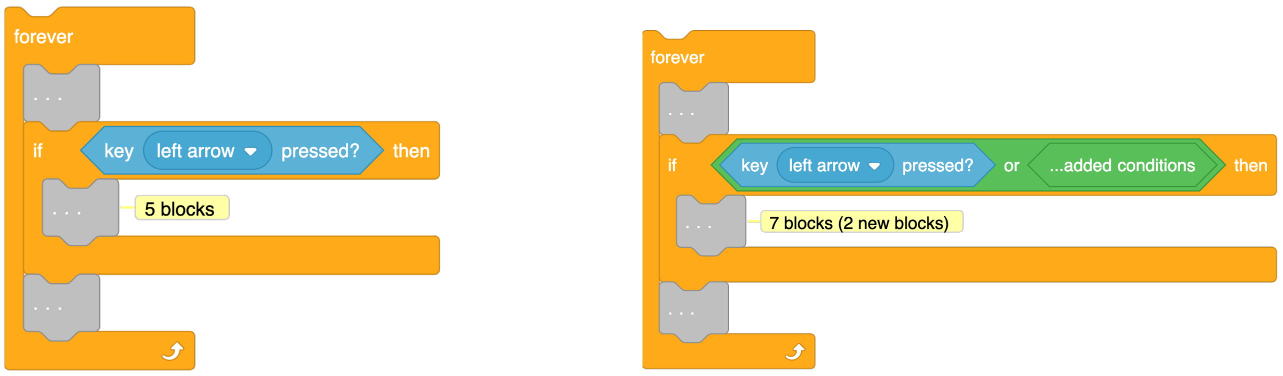

Figure 10. Insertion example for Forever IF/IF Else in original and its remix

Operation	% of change of line
Delete	81.66%
Update	13.58%
Insert	4.76%

Table 4. Prevalence of operations found in Nested IF Else

Operation	% of change of line
Delete	53.95%
Insert	26.55%
Update	19.50%

Table 5. Prevalence of operations found in Forever IF/IF Else

6 Discussion

In this section, we discuss how the prevalence of programming idioms and code changes in the remixes may be leveraged by computing learners and educators.

6.1 Identifying Scratch Idioms

Similarly to prior approaches, we found the task of identifying common Scratch programming idioms quite challenging. What comprises an idiom can be a highly subjective question. The issue at hand is to determine the prevalence threshold, exceeding which an idiom would be considered for inclusion. Some idioms prominently featured in language tutorials may not manifest themselves prominently in actual application codebases. In fact, a related work effort reports a similar observation regarding a different set of idioms [10]. Their results show that the top three recurring patterns appear in only 19% to 34% of over 200k project samples, while the rest of the patterns appear in less than 10% of the samples.

This insight suggests that the Scratch application codebase is rife with non-idiomatic coding practices. One possible explanation for this insight is that Scratch programmers tend to code in more experimental styles typical of bottom-up programming practices, more concerned with the end result rather than with the specifics and quality of their code.

This phenomenon also reflects how workarounds similar to those documented in Scratch's learning resources [4] may be commonly found among projects created by novice programmers. To identify the common programming idioms as they pertain to actual programming practices, one might explore applying the software mining techniques [9] on the actual application codebases rather than systematically consulting textbooks and tutorials.

6.2 Programming Idioms in Novice Programming Practices

Our findings suggest that novice programmers can successfully recognize and make use of programming idioms. Some idioms are highly prevalent (i.e., Change And Wait, Repeat Change, and Forever IF/IF Else), while several others (e.g., Iterate List, Repeat Ask, etc.), not so much. The low prevalence of many idioms may be due to their more specific use cases, and advanced usage scenarios. Nevertheless, the presence of the observed highly recurring idioms suggests that novice programmers, similar to professional programmers, find idioms natural to learn and apply in their coding practices. A similar observation has been discussed in prior work about the naturalness of event-based programming styles among Scratch novice programmers [21].

Although highly recurring idioms are applicable regardless of the project categories, many of the studied idioms are often domain-specific. Any Scratch project has at least some graphics, making animation-related idioms (i.e., Change And Wait, Repeat Change) highly prevalent. However, some idioms are more prevalent within certain project categories. For example, Forever IF/IF Else in the games and tutorials category represent more than 50% of all projects and tend to be less common in the art category (~23%).

Tutorial projects present an interesting case, as they seem to reflect the average of all categories. The likely impetus for creating these projects is the natural desire of programmers to share their knowledge and expertise with their fellow programmers. It is perhaps for this reason, that tutorial projects often contain the idioms commonly used across all categories (e.g., art, game, storytelling) that we studied.

6.3 Programming Idioms and Introductory Computing

The low-prevalent idioms suggest possible opportunities for educational interventions. For example, the fact that ITERATE LIST is not commonly present suggests that list data structures are underused in the application codebase. Perhaps introductory learners are unaware of this data structure and its common applications. Hence, it might be more effective to teach programming constructs in terms of their relevant idioms, so learners can quickly start becoming familiarized with the usage of programming constructs and apply them appropriately in their programming practices. In the case of the list data structure, educators can more explicitly introduce the list idioms as part of tutorials and sample projects to promote the usage of this important data structure in introductory programming. Similarly, REPEAT ASK, used for validating text, a common idiom in other programming languages for securing the program input. Perhaps better familiarity is all that is required for not only using the *ask* and *answer* blocks, but also for the practice of validating input becoming a standard tool for Scratch programmers.

6.4 Roles of Programming Idioms in Remixing Practices

Khawas et al.[23] explored the overall changes between the projects and their remixes. They found that programmers often insert blocks than delete them. However, the insertions exceed deletions only by a small margin. In our work, we also made it a point to study how the code fragment within the idiom's body changes between projects and their remixes. Specifically, we determined that in two common flow control idioms, programmers more frequently delete than insert code blocks in the remixes (i.e., 81.66% in NESTED IF ELSE, 53.95% in FOREVER IF/IF ELSE).

From Tables 5 and 4, the insertion in NESTED IF ELSE is lower than in FOREVER IF/IF ELSE, and the potential reason for this discrepancy is that NESTED IF ELSE contains specific operations in potential scenarios, which programmers usually keep or delete as a whole. The FOREVER IF/IF ELSE idiom allows more editing flexibility, so programmers tend to actually insert code blocks into this idiom. The dissimilar percentages of code insertions between the two common flow-control idioms raise an interesting question: do these differences correlate with the degree of cognitive effort required to understand each idiom?

As we determined, the majority of changes for these two idioms in the remixes are deletions. This observation suggests that programmers remix projects by retaining existing control structures, from which they replace much of the contained code blocks. Driven by this observation, one may consider providing students with generic forms or examples of common idioms (i.e., skeletal idiom code). Then one can guide students to fill in the necessary logic, so as to more effectively master these idioms. This observation may present opportunities for educators and designers of language learning environments.

7 Threats to validity

The validity of our analysis results may be endangered by a few factors. The programming idioms we documented and studied are limited by the Scratch programming materials available at the time of the study. New prevailing idioms may be uncovered with any changes in the language features, the community's programming practices or educational interventions in this language.

To ensure the accuracy of our idiom detection and to avoid introducing false positives, we implemented a suite of test cases for each detection algorithm. We also implemented more generalized detectors to detect idioms with certain variants.

To investigate changes within idioms, we only selected two most common control flow idioms, as their structures reliably isolate the differences between the original projects and their remixes, so we could precisely measure these differences. Hence, our findings may not be representative of the actual changes across all common idioms introduced by programmers in their remixes.

8 Conclusion and Future Work

Our work sheds light on common Scratch programming idioms. Our large-scale study experiment assesses not only the prevalence of Scratch programming idioms, but also how programmers tend to change code within two common idioms between the remixed projects and their sources. The results of our work identify common programming idioms, a piece of knowledge that can benefit novice programmers as a way to help them learn the language faster. The net effect would be promoting effective programming practices among introductory learners, writing idiomatic high quality code.

Possible future work directions include exploring common programming idioms in this domain from the perspective of novice programmers as well as applying software mining and natural language techniques to extract idioms from existing application codebases[9].

Acknowledgments

The authors would like to thank the anonymous reviewers and our shepherd, Meghan Allen, for their valuable feedback that helped improve this manuscript. This research was supported in part by NSF through the grants #1744722 and 1712131.

References

[1] 2021. Example for iterating the list. https://programming-idioms.org/idiom/6/iterate-over-list-values.

[2] 2021. Example for removing elements from the list. https://programming-idioms.org/idiom/136/remove-all-occurrences-of-a-value-from-a-list.

[3] 2021. Forever If (). https://en.scratch-wiki.info/wiki/Forever_If_()_(block).

[4] 2021. List of Block Workarounds. https://en.scratch-wiki.info/wiki/List_of_Block_Workarounds.

[5] 2021. Online programming idioms. https://programming-idioms.org/.

[6] 2021. Scratch's statistics. https://scratch.mit.edu/statistics/.

[7] Efthimia Aivaloglou and Felienne Hermans. 2016. 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*. 53–61.

[8] Carol V Alexandru, José J Merchante, Sebastiano Panichella, Sebastian Proksch, Harald C Gall, and Gregorio Robles. 2018. On the usage of pythonic idioms. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. 1–11.

[9] Miltiadis Allamanis and Charles Sutton. 2014. Mining idioms from source code. In *Proceedings of the 22nd acm sigsoft international symposium on foundations of software engineering*. 472–483.

[10] Kashif Amanullah and Tim Bell. 2018. Analysing students' scratch programs and addressing issues using elementary patterns. In *2018 IEEE Frontiers in Education Conference (FIE)*. IEEE, 1–5.

[11] Kashif Amanullah and Tim Bell. 2019. Analysis of progression of scratch users based on their use of elementary patterns. In *2019 14th International Conference on Computer Science & Education (ICCSE)*. IEEE, 573–578.

[12] Kashif Amanullah and Tim Bell. 2019. Evaluating the use of remixing in scratch projects based on repertoire, lines of code (loc), and elementary patterns. In *2019 IEEE Frontiers in Education Conference (FIE)*. IEEE, 1–8.

[13] Kashif Amanullah and Tim Bell. 2020. Teaching Resources for Young Programmers: the use of Patterns. In *2020 IEEE Frontiers in Education Conference (FIE)*. IEEE, 1–9.

[14] A. Anthropy. 2019. *Make Your Own Scratch Games!* No Starch Press. https://books.google.com/books?id=QVv6DwAAQBAJ

[15] Owen Astrachan and Eugene Wallingford. 1998. Loop patterns. In *Proc. Fifth Pattern Languages of Programs Conference, Allerton Park, Illinois*.

[16] J. Bergin. 1999. Pattern for selection version 4. https://csis.pace.edu/~bergin/patterns/.

[17] Michael J Clancy and Marcia C Linn. 1999. Patterns and pedagogy. *ACM SIGCSE Bulletin* 31, 1 (1999), 37–42.

[18] Deborah A. Fields, Michael Giang, and Yasmin Kafai. [n.d.]. Programming in the Wild: Trends in Youth Computational Participation in the Online Scratch Community. In *Proceedings of the 9th Workshop in Primary and Secondary Computing Education* (New York, NY, USA, 2014-11-05) *(WiPSCE '14)*. Association for Computing Machinery, 2–11. https://doi.org/10.1145/2670757.2670768

[19] G. Ford, M. Ford, and S. Ford. 2017. *Hello Scratch!: Learn to Program by Making Arcade Games*. Manning Publications. https://books.google.com/books?id=Te4jvgAACAAJ

[20] Christoph Frädrich, Florian Obermüller, Nina Körber, Ute Heuer, and Gordon Fraser. 2020. Common bugs in scratch programs. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*. 89–95.

[21] Michal Gordon, Assaf Marron, and Orni Meerbaum-Salant. 2012. Spaghetti for the main course? Observations on the naturalness of scenario-based programming. In *Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education*. 198–203.

[22] Paul A hner, John Sweller, and Richard E Clark. 2006. Why Minimal Guidance During Instruction Does Not Work: An Analysis of the Failure of Constructuvist, Discovery, Problem-Based, Experimental, and Inquiry-Based Teaching. *Educational Psychologist* 42, 2 (2006).

[23] Prapti Khawas, Peeratham Techapalokul, and Eli Tilevich. 2019. Unmixing remixes: The how and why of not starting projects from Scratch. In *2019 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 169–173.

[24] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. 2010. The scratch programming language and environment. *ACM Transactions on Computing Education (TOCE)* 10, 4 (2010), 1–15.

[25] M. Marji. 2014. *Learn to Program with Scratch: A Visual Introduction to Programming with Games, Art, Science, and Math*. No Starch Press. https://books.google.com/books?id=sYvlAwAAQBAJ

[26] Howard A Peell. 1987. An APL idiom inventory. In *Proceedings of the international conference on APL: APL in transition*. 362–368.

[27] Alan J Perlis and Spencer Rugaber. 1979. Programming with idioms in APL. *ACM SIGAPL APL Quote Quad* 9, 4-P1 (1979), 232–235.

[28] Michael Smit, Barry Gergel, and H James Hoover. 2011. Code convention adherence in evolving software. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 504–507.

[29] A. Sweigart. 2021. *Scratch 3 Programming Playground: Learn to Program by Making Cool Games*. No Starch Press. https://books.google.com/books?id=L967DwAAQBAJ

[30] Peeratham Techapalokul and Eli Tilevich. 2017. Understanding recurring quality problems and their impact on code sharing in block-based software. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 43–51.

[31] E.A. Vlieg. 2016. *Scratch by Example: Programming for All Ages*. Apress. https://books.google.com/books?id=kxoPDQAAQBAJ

[32] David Weintrop and Uri Wilensky. 2015. To block or not to block, that is the question: students' perceptions of blocks-based programming. In *Proceedings of the 14th international conference on interaction design and children*. 199–208.