International Journal of Computer Science Education in Schools, August 2018, Vol. 2, No. 3
ISSN 2513-8359

Computational Concepts Reflected on Scratch Programs

Kyungbin Kwon[1]
Sang Joon Lee[2]
Jaehwa Chung[3]

[1]Indiana University
[2]Mississippi State University
[3]Korea National Open University

DOI:

Abstract

Evaluating the quality of V W X G H Q W V ¶ S U R J U D P s is necessary for better teaching and learning. Although many innovative learning environments for computer science have been introduced, the scarcity of program evaluation frames and tools is a demanding issue in the teaching practice. 7 K L V V W X G \ H [D P L Q H G W K H T X D O L W \ R I V W X G
utilizing Dr. Scratch and by analyzing codes based on four computational concepts: conditions, loops, abstractions, and variables. Twenty-three Scratch programs from two classes of pre-service teachers from a university were examined. Dr. Scratch results revealed that Scratch programs demonstrated a middle level of competency in computational thinking. The analysis of computational concepts suggested that students had a sufficient understanding of the main concepts and demonstrated computing competency by applying the concepts into their programs. The study also discussed inefficient programming habits, runtime issues, utilizing Scratch, and the importance of problem decomposition skills

Keywords: Scratch, block-based programming, computer science education, novice programmer, computational thinking

1. Introduction

Since President Obama addressed the importance and urgency of Computer Science (CS) education in K-12, many stakeholders including education administrators, scholars, teachers, and government agencies, such as NSF, have tried to develop a sustainable curriculum that encourages more students to learn to program earlier. However, the deficiency of CS education in K-12 is not getting better in the US. Although nine out of ten parents surveyed want their children to learn computer science, only 40% of middle and high schools teach computer programming (Google & Gallup, 2015).

It is well known that there are several barriers to overcome, such as 1) insufficient professional development for in-service teachers (Buss & Gamboa, 2017; Reding et al., 2016), V W X G H Q W V ¶ Q H J D W L Y H D W W L W X G
self-efficacy toward CS (Arraki et al., 2014; Google & Gallup, 2017; Simsek, 2011), and 3) the lack of evidence proving the effect of teaching practice utilizing innovative learning environments, such as code.org and Scratch
. D O H O L R ÷ O X   * • O E D -K H \ U Q   5 R E O H V * R Q] R P H].

To resolve the issues, nowadays, many schools have been suggested to adopt block-based programming (BBP) environments, for example, Scratch (https://scratch.mit.edu/) where V W X G H Q W V G H Y H O R S D S U R J U D P
V Q D S S L Q J E O R F N V ´ 5 H V H D U F K H U V K D Y H S U R Y H G W K D W W K H % % 3 P D
providing recognizable commands in a block form. It also eliminates syntactic errors by constraining the structures of a program using different shapes and combining commands into chunks (Bau, Gray, Kelleher, Sheldon, & Turbak, 2017).

Although many teachers and researchers have introduced Scratch to their classrooms, not many attempts have been made to evaluate the quality of Scratch programs and provide tailored feedback to students based on the results. There is a common question many teachers want to answer: How can I assess the quality of students' Scratch programs?

As Chao (2016) suggested, we can assess computational concepts (conditions for decision-making, iterations with specified cycle, data representation, etc.), computational designs (decomposition of problems, sequences of tasks, etc.), and computational performances (identification of goals, optimization of programs, usability, etc.) to evaluate students' programming competency and strategies reflected on Scratch programs. The size and complexity of a program can be the quantitative indicators of Scratch programs (Aivaloglou & Hermans, 2016). Utilizing a web tool like Dr. Scratch also enables us to evaluate Scratch programs automatically (Moreno-León, Robles, & Román-González, 2015).

Although these evaluation concepts and methods are available, the lack of an assessment rubric or evaluation frame results in the scarcity of code evaluations in the teaching practice (Moreno-León et al., 2015). There is also a need for studies exploring the validity of the assessment across the evaluation tools (Buffum et al., 2015; Grover & Pea, 2013). The need to examine the characteristics of Scratch programs has increased because it will reveal the status of students' computational thinking (Moreno-León et al., 2015).

The primary purpose of the current study is to examine the quality of students' Scratch programs by utilizing Dr. Scratch and by analyzing codes based on computational concepts. The close evaluation of Scratch programs will reveal weak areas that students struggle in and provide instructional insight to design learning activities. The following research questions were addressed:
Q1. What was the general quality of students' Scratch programs based on Dr. Scratch's evaluation?

Q2. What computational concepts were reflected on students' Scratch programs?

2. Literature Review

2.1 Block-based programming

Novice programmers often lose their cognitive capacity while figuring out the surface features of programming, such as syntax rules, and easily fail to apply programming concepts to develop effective solutions (Lahtinen, Ala-Mutka, & Järvinen, 2005; Winslow, 1996). Considering the limitation of novice programmers, BBP excludes the chances of syntactical errors, uses commands similar to spoken languages, provides immediate feedback, and visualizes abstract concepts, such as variables, which reduces the cognitive load (Maloney, Resnick, Rusk, Silverman, & Eastmond, 2010). These features of BBP allow novice programmers to grab the fundamental programming concepts easily (Buitrago Flórez et al., 2017). BBP also provides novice programmers with "fun" components by allowing them to create authentic programs, such as games, interactive stories, and animations that demonstrate their problem solving skills (Resnick et al., 2009).

Since BBP provides a visual programming environment, which is suitable for teaching programming concepts, the use of BBP has increased in introductory programming education courses (Aivaloglou & Hermans, 2016). Scratch is one of the most commonly used BBP and provides a media-rich interactive programming environment. Developed by the MIT Media Lab, Scratch was intended to make programming accessible and engaging for everyone (Resnick et al., 2009). With Scratch, not only is it easy for people with limited or no programming background to begin learning programming concepts, but it is also possible to create increasingly complex programs over time (Sáez-López, Román-González, & Vázquez-Cano, 2016; Su, Yang, Hwang, Huang, & Tern, 2014). Because of its visual nature and an intuitive drag and drop method of programming, Scratch is ideal for young people and expected to be a potential language for K-12 computer science (Sáez-López et al., 2016). The visual programming allows young students to create scripts easily by playing and interacting with blocks. While working on interactive stories, games, and animations individually and collaboratively with peers, users are able to learn mathematical and computational concepts as well as 21st century skills, including critical thinking, creativity, communication, and collaboration (Maloney et al., 2010; Resnick et al., 2009).

BBP has enabled computer science educators to implement computational problem-solving (e.g., Liu, Cheng, & Huang, 2011; Topalli & Cagiltay, 2018). Computational problem-solving integrates real-life issues, which students can solve by developing a program. Because computer science requires problem-solving skills for broad issues, computational problem-solving is one of the core competencies of computer science (Liu et al., 2011). It is also aligned with the current CS education paradigm of "teaching computer science in context" to encourage students to learn computing in a concrete and personal way (Cooper & Cunningham, 2010). The benefits of computational problem-solving include (1) an enhanced understanding of programming concepts, logic, and computational practices (Sáez-López et al., 2016), (2) better performances on designing software system (Topalli & Cagiltay, 2018), and (3) decomposition of computational problems and adoption of design strategies (Chao, 2016).

2.2 Evaluation of program competence

2.2.1 Dr. Scratch

Moreno-León et al. (2015) introduced Dr. Scratch (http://www.drscratch.org): a web application that analyzes Scratch programs. Dr. Scratch evaluates student's computational-thinking competence based on seven criteria: abstraction and problem decomposition, logical thinking, synchronization, parallelism, algorithmic notions of flow control, user interactivity, and data representation. When users submit their Scratch programs, Dr. Scratch displays numeric scores of the criteria (zero to three) as well as the overall level of mastery in terms of basic, developing, and master. By utilizing Dr. Scratch, students as well as instructors can easily evaluate their Scratch programs and get immediate feedback (see Moreno-León et al. (2015) paper for more information regarding the rubric of the assessment.).

2.2.2 Computational concepts

Although Dr. Scratch provides quantitative scores, it is not enough to evaluate students' understanding of computational concepts. Developing Scratch programs involves several computational concepts. For example, making a sprite move predetermined paths repeatedly until a particular event occurs requires the understanding of loops and conditions at least. In other words, Scratch allows students to demonstrate the understanding of computational concepts through their programs and learning activities (Grover, Pea, & Cooper, 2015; Lee, 2010). Main computational concepts include loops, conditions, sequence, event handling, Boolean logic, variables, message passing, algorithmic flow of control, problem decomposition, abstraction, and so on.

Although all of the concepts are crucial and interact with each other for a program, the current study focuses on four main concepts (loops, conditions, variables, and abstraction) that novice programmers easily find difficulty understanding and applying to their Scratch programs (Grover et al., 2015; Kwon, 2017; Shi, Cui, Zhang, & Sun, 2018). While many students understand the concept of simple loops, for example, they struggle with loops that involve variables (Grover & Basu, 2017). Usually, loops need to be specified how many times instructions will run or when the repeat will stop. This specification involves arithmetic or conditional expressions integrating variables. In some cases, the value of variable changes as the loop runs, which students often misunderstood (Grover & Basu, 2017).

As Wing (2006, p. 35) emphasized while defining computational thinking, thinking like a computer scientist requires "thinking at multiple levels of abstraction". By using the concept of abstraction, students can decompose a complex problem into manageable steps and modularize solutions (Wing, 2006). Thus, students can generalize and transfer a solution to other similar problems when they use the power of abstraction (Yadav, Hong, & Stephenson, 2016). Scratch allows students to define their own blocks, which enables them to customize complex codes into a "reusable" block. Although this method demonstrates the power of abstraction, students often fail to use the user-defined blocks (Moreno & Robles, 2014).

There are several trials to measure students' computational thinking through tests (e.g., Grover & Basu, 2017; Meerbaum-Salant, Armoni, & Ben-Ari, 2013). Considering that understanding concepts is necessary but not sufficient to develop an effective program, analyzing students' programs is crucial in evaluating the internalization of computational concepts (Arzarello, Chiappini, Lemut, Malara, & Pellerey, 1993). In this sense, the current study aimed to reveal students' computational concepts by analyzing their Scratch programs.

3. Method

3.1 Participants

Twenty-three students were recruited from two of the same undergraduate courses offered in a large Midwest university in the US. They were pre-service teachers taking the Computer Educator License (CEL) program in addition to their major. The majority of participants were female (21) with no or at most little computer programming experience before taking the course. They were not given compensation for their participation in the study. The study was approved by the University Institutional Review Board (#1701722036).

3.2 Context of Learning

The final goal of the course was to train pre-service teachers to develop simple programs to solve authentic problems, such as printing a receipt for cashiers, calculating tips in a restaurant, and creating a quiz. The students learned the basic concepts of programming and syntax of Python throughout a semester and developed the programs as the final project. At the early period of instruction (Week 2~7), the instructor utilized Scratch to let students practice programming concepts without being distracted by syntax issues. In Week 7, students submitted their Scratch programs as the midterm projects that were the artifacts we analyzed in this study.

Students were required to use variables, conditional blocks, repeating blocks, and user-defined blocks. The instructor also emphasized the importance of the logical procedure of computing. Students selected their program topic.

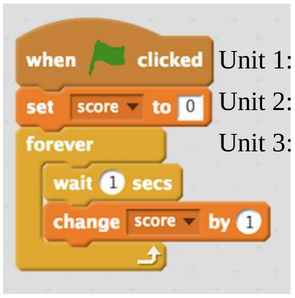
3.3 Data

Students' Scratch programs were collected and the files were uploaded to researcher's Scratch account to be analyzed. We captured the screens of each individual sprite's code and saved it to image files for analysis. A total of 23 programs were collected.

3.4 Identifying the quality of programs

To evaluate the quality of Scratch programs, we utilized Dr. Scratch and analyzed code manually. As discussed, Dr. Scratch provides scores (0 to 3) regarding the level of seven computational thinking competencies, which ranges 0 to 21. The current study presents the quantitative results of Scratch programs to describe the overall quality.

To have an in-depth understanding of students' Scratch programs, we analyzed them, while considering computational concepts: conditions, loops, abstractions, and variables. The first author analyzed the programs and the other authors validated the analysis. Different interpretations among the authors were resolved after negotiating. The unit of analysis was a semantic unit including one or several code blocks that executed a certain task. For example, the following code represents three semantic units (see Figure 1).



```
when [flag] clicked
set score to 0
forever
    wait 1 secs
    change score by 1
```

Unit 1: Event triggering the following instructions
Unit 2: Setting the value (0) to the variable (score)
Unit 3: Repeating the instructions "forever"

Figure 1. Example of the units of analysis

4. Results

In the following sections, the analysis of Scratch programs is presented. First, the results of Dr. Scratch evaluation suggest the overall quality of the programs. Then, the manual analysis of codes identifies the strengths and areas for improvement regarding computational concepts reflected on the programs.

4.1 Overview of programs

Table 1 describes the descriptive information of Scratch programs organized by its types. There were two types of programs: game and quiz. Because the types of programs might affect the number of sprites and user interactivities, we categorized programs according to the types and coded them accordingly. Overall, games, except drawing, added multiple sprites that included a main character and some obstacles. In contrast, quizzes involved fewer sprites mainly asking and answering questions and some objects indicating correct answers (Q03) or levels of difficulty (Q04). Except for a few programs, most used only one backdrop, which suggests they were mostly made up of one mission or one theme. The number of block clusters (sets of blocks attached together) and block codes (individual blocks) indicated the complexity of the programs. Relatively, quizzes created more clusters of blocks as well as blocks. It is noteworthy that there are considerable variations among programs regarding the number of clusters and individual blocks developed. The higher number of clusters and blocks implies the higher complexity of the program in this study. We also found significant positive correlations between the scores evaluated by Dr. Scratch and the number of clusters, *r(22)* = .53, *p* = .01 and the number of blocks *r*(22) = .56, *p* = .005.

Table 1

Descriptive information of programs by type

Type	Code	# Sprites	# Backdrops	# Block Clusters	# Block Codes	Dr. Scratch Score
GAME		4.7	1.4	14.3	70.2	13.2
Maze	G01	7	1	7	55	12
	G02	5	1	8	46	12
	G03	4	1	14	49	12
	G04	2	1	7	28	9
	G05	5	1	24	90	15
	G06	10	5	50	191	17
Catch	G07	5	1	15	105	16
	G08	4	1	7	46	13
Dance	G10	4	1	7	46	11
Drawing	G09	1	1	4	46	15
QUIZ		4.0	1.8	20.7	114.2	13.8
Quiz	Q01	2	1	5	47	14
	Q02	2	1	11	75	13
	Q03	7	2	14	129	15
	Q04	16	3	101	419	15
	Q05	3	1	6	73	14
	Q06	1	1	3	70	14
	Q07	2	1	5	62	13
	Q08	4	1	9	53	15
	Q09	1	1	2	29	9
	Q10	1	1	2	45	11
	Q11	1	1	6	62	12
	Q12	4	1	21	121	17
	Q13	8	8	84	299	17
TOTAL		4.3	1.6	17.9	95.0	13.5

Note: G stands for game and Q does quiz in the code. # stands for number of.

4.2 Dr. Scratch evaluations

The evaluations of Dr. Scratch revealed that students demonstrated the middle level of competence (Total score =13.5 out of 21, the average score of all criteria = 1.9 out of 3). Regarding individual criteria, programs showed similar results on data representation, abstraction, interactivity, synchronization, and logic. All programs, except one, demonstrated higher than level 2 on the flow control. Regarding the parallelism, while seven programs received level 3, twelve received level 0 or 1, which showed considerable variations among programs. Regarding the overall level of competence, 14 programs received the developing level, and the rest of the projects earned the master level.

Table 2

Descriptive statistics of Dr. Scratch evaluations of programs

	Overall score	Flow	Data	Abstraction	Interactivity	Synchronization	Parallelism	Logic
Mean	13.5	2.4	2.0	2.0	2.0	1.8	1.4	1.9
SD	2.31	0.58	0.21	0.00	0.00	1.11	1.16	0.73

Note: N = 23

4.3 Analysis of Scratch codes

In the following sections, students' computational concepts reflected on the Scratch programs are presented. The descriptive analysis of computational concepts aimed to reveal students' computing competency focused on conditions, loops, abstractions, and variables.

4.4 Conditions

To make a decision, students need to consider conditions. The condition is integrated into if-blocks as well as repeat-blocks in Scratch. In this section, we examine only if-blocks. The if-blocks can be specified into three types: (1) simple if-block that considers only one condition (if), (2) if-else block that considers two conditions: true and an alternative, and (3) nested if-block that considers multiple conditions by integrating multiple if or if-else blocks.

In developing Scratch programs, students need to define conditions logically so they can tell in which condition a particular instruction will execute. The condition can be expressed with an event, such as touching, or with multiple operators, such as logical expression and arithmetic comparisons. The ability to utilize if-blocks is related to the competence of logical thinking. We examined students' Scratch programs based on the structure of if-blocks and their conditional statements.

4.4.1 Students seemed to use if-blocks properly according to the purpose of program.

G01, for example, used if-blocks to see whether a sprite "touch" a line or obstacles (other sprites). So the structure was simple as "if touching color or sprite then." G01 also used nested if-blocks to consider three conditions: age = 25, age < 25, and the other condition (age > 25). The structure was logically valid and clear to make the decision (see Figure2-a).

Q04 developed a nested if-block to control the flow of the program. Q04 asked questions and updated scores according to the answers. The if-block added (or subtracted) scores once the answer was correct (or incorrect) and switched the level of difficulty once the score reached a certain point.

Q05 developed a quiz where a sprite tried to catch up with a shark. When a user answered a question correctly, the sprite reduced the distance with the shark and vice versa. Q05 also used the nested if-blocks to decide the flow of the program according to the distance and the user's intention to beat the shark as illustrated in Figure2-b. Q05 demonstrated an effective way to use if-blocks to control the program flow based on multiple conditions.

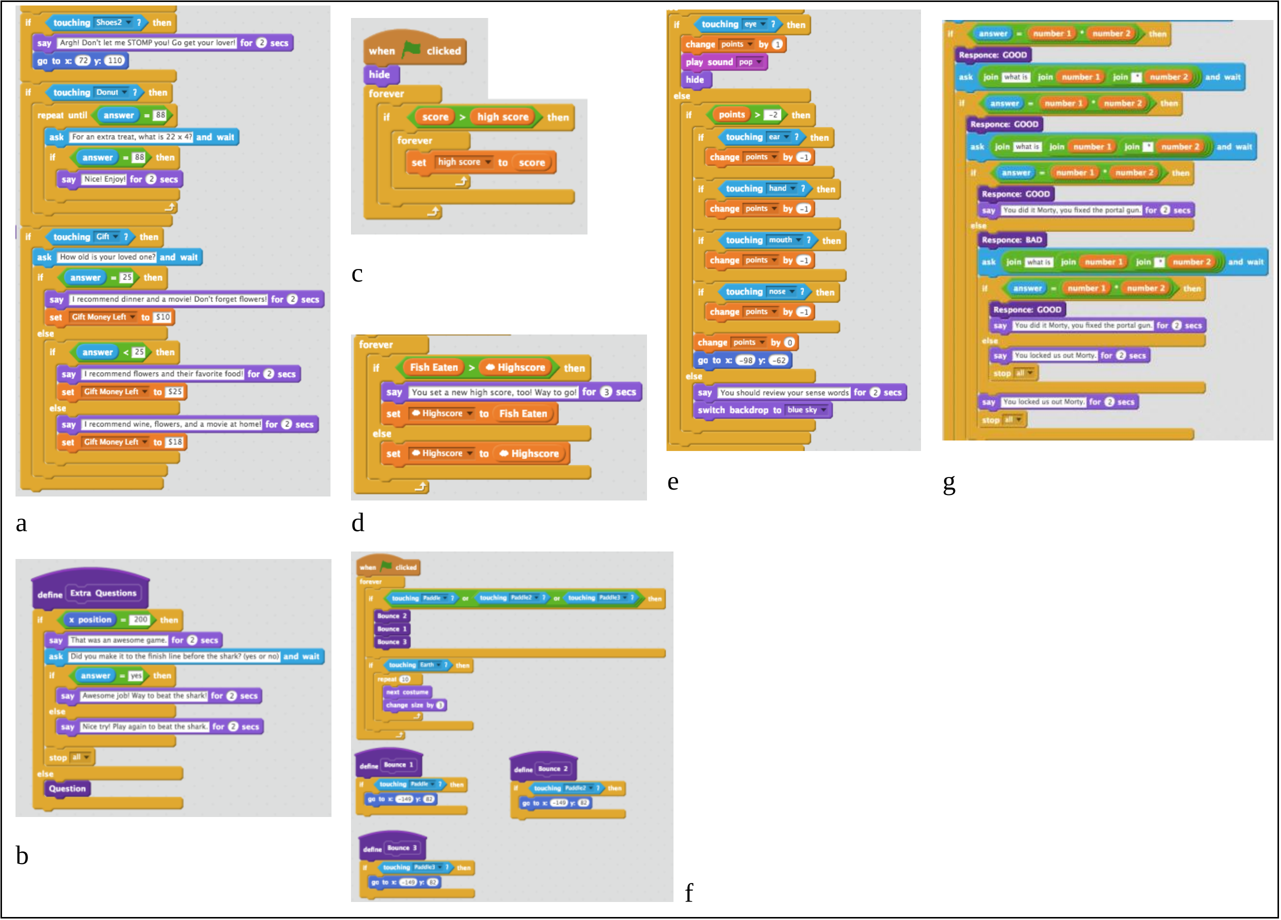
Q08 updated the "high score" using the if-block as Figure2-c. The code demonstrated how the student updated the value in consideration of the condition. (Please note that the forever-block inside the if-block was not necessary. We will discuss it later.)

All programs expressed conditional statements appropriately, which demonstrated that students fully understood how to develop the conditional expressions.

4.4.2 Students considered an alternative condition that was not necessary.

G07 considered the highest score users got and updated its value whenever it was broken (see Figure2-d). The first if part compared the current user score (Fish Eaten) and the highest score (Highscore) and updated the Highscore with the Fish Eaten in case the Fish Eaten is bigger than the Highscore. The following else part, however, updated Highscore with the same value (Highscore). The update with the value of itself was unnecessary in that context.

Q04 created a quiz that asked users to match words and pictures. Q04 used nested if-blocks to check correct matches first and incorrect matches in the else structure as nested (see Figure2-e). However, the alternative matches were all the incorrect answers and were not necessary to be specified. This suggests that students need to be trained to decide which condition should be defined and which conditions can be treated as an alternative without the specification.



a
```
if <touching Shoes2 ?> then
  say Argh! Don't let me STOMP you! Go get your lover! for 2 secs
  go to x: 72 y: 110
if <touching Donut ?> then
  repeat until <answer = 88>
    ask For an extra treat, what is 22 x 4? and wait
    if <answer = 88> then
      say Nice! Enjoy! for 2 secs
if <touching Gift ?> then
  ask How old is your loved one? and wait
  if <answer = 25> then
    say I recommend dinner and a movie! Don't forget flowers! for 2 secs
    set Gift Money Left to $10
  else
    if <answer < 25> then
      say I recommend flowers and their favorite food! for 2 secs
      set Gift Money Left to $25
    else
      say I recommend wine, flowers, and a movie at home! for 2 secs
      set Gift Money Left to $18
```

b
```
define Extra Questions
if <x position = 200> then
  say That was an awesome game. for 2 secs
  ask Did you make it to the finish line before the shark? (yes or no) and wait
  if <answer = yes> then
    say Awesome job! Way to beat the shark! for 2 secs
  else
    say Nice try! Play again to beat the shark. for 2 secs
  stop all
else
  Question
```

c
```
when clicked
hide
forever
  if <score > high score> then
    forever
      set high score to score
```

d
```
forever
  if <Fish Eaten > Highscore> then
    say You set a new high score, too! Way to go! for 3 secs
    set Highscore to Fish Eaten
  else
    set Highscore to Highscore
```

e
```
if <touching eye ?> then
  change points by 1
  play sound pop
  hide
else
  if <points > -2> then
    if <touching ear ?> then
      change points by -1
    if <touching hand ?> then
      change points by -1
    if <touching mouth ?> then
      change points by -1
    if <touching nose ?> then
      change points by -1
    change points by 0
    go to x: -98 y: -62
  else
    say You should review your sense words for 2 secs
    switch backdrop to blue sky
```

f
```
when clicked
forever
  if <<touching Paddle ?> or <touching Paddle2 ?> or <touching Paddle3 ?>> then
    Bounce 2
    Bounce 1
    Bounce 3
  if <touching Earth ?> then
    repeat 10
      next costume
      change size by 3

define Bounce 1
if <touching Paddle ?> then
  go to x: -149 y: 82

define Bounce 2
if <touching Paddle2 ?> then
  go to x: -149 y: 82

define Bounce 3
if <touching Paddle3 ?> then
  go to x: -149 y: 82
```

g
```
if <answer = number 1 * number 2> then
  Responce: GOOD
  ask join what is join number 1 join * number 2 and wait
  if <answer = number 1 * number 2> then
    Responce: GOOD
    ask join what is join number 1 join * number 2 and wait
    if <answer = number 1 * number 2> then
      Responce: GOOD
      say You did it Morty, you fixed the portal gun. for 2 secs
    else
      Responce: BAD
      ask join what is join number 1 join * number 2 and wait
      if <answer = number 1 * number 2> then
        Responce: GOOD
        say You did it Morty, you fixed the portal gun. for 2 secs
      else
        say You locked us out Morty. for 2 secs
        stop all
    say You locked us out Morty. for 2 secs
    stop all
```

Figure 2. Codes representing the concepts of conditions

4.4.3 A student considered the same conditions twice unnecessarily.

G02 created a game: "Traveling without touching obstacles." The first if-block checked an occasion when a sprite touched the obstacles (Paddle, Paddle2, and Paddle3) as Figure2-f illustrates. A close examination of the user-defined blocks revealed that the three if-blocks rechecked the "touching" condition that could be removed, so the hosting if-block checked three conditions at once.

4.4.4 A student used the nested if-blocks inefficient way.

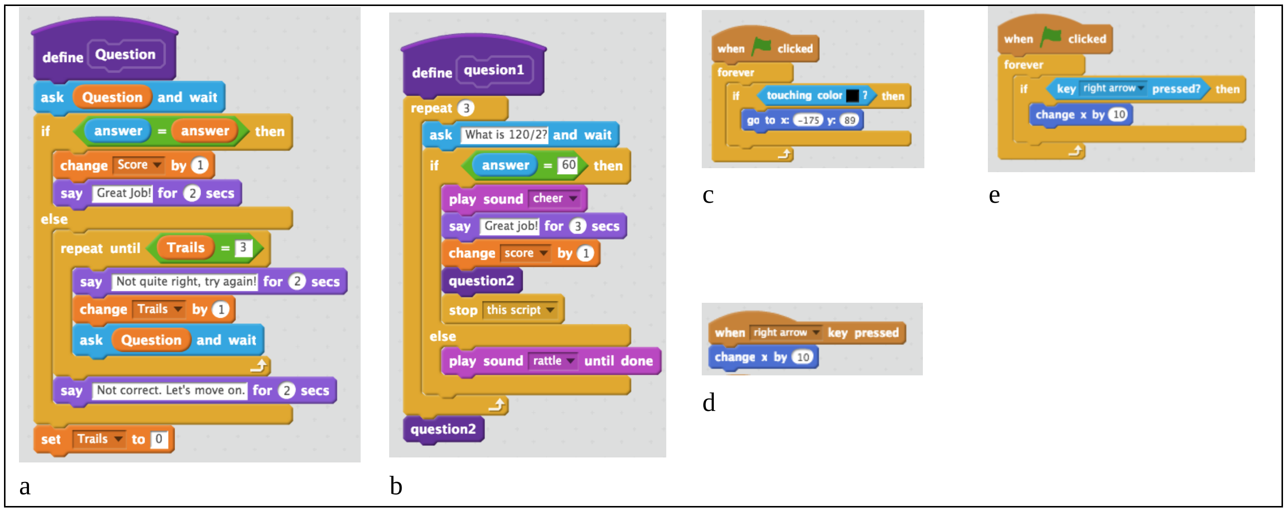
Q07 nested multiple conditions in order. It seemed the student developed the nested if-block as she/he drew a decision tree as Figure2-g. However, Q07 solution made the code complex and difficult to trace. Because the results of the conditions were independent, there was no reason for nesting multiple conditions. It seemed the students did not fully understand the logic of if-blocks and simply followed the decision tree process.

4.4.5 Usages of if-blocks

While most game programs used the simple if-blocks (8 out of 10), most quiz programs used the if-else or nested if-blocks (11 out of 13). It suggests that students selected the types of conditions according to the purpose of programs, such as checking correct answers that required at least if-else blocks or touching objects that used simple if-blocks.

4.5 Loops

In order to use a loop properly, students need to identify which instructions repeatedly run until a certain condition is reached. Thus, the purpose of a loop and a condition to stop the loop are critical elements to evaluate its effectiveness. In Scratch, there are three types of loops: forever (repeat without stop), repeat <times>, and repeat until <condition>. The ability to utilize the repeat block is related to the competence of flow control.



```
define Question
ask Question and wait
if answer = answer then
    change Score by 1
    say Great Job! for 2 secs
else
    repeat until Trails = 3
        say Not quite right, try again! for 2 secs
        change Trails by 1
        ask Question and wait
    say Not correct. Let's move on. for 2 secs
set Trails to 0
```

```
define quesion1
repeat 3
    ask What is 120/2? and wait
    if answer = 60 then
        play sound cheer
        say Great job! for 3 secs
        change score by 1
        question2
        stop this script
    else
        play sound rattle until done
question2
```

```
when clicked
forever
    if touching color ? then
        go to x: -175 y: 89
```

```
when right arrow key pressed
change x by 10
```

```
when clicked
forever
    if key right arrow pressed? then
        change x by 10
```

Figure 3. Codes representing the concepts of loops

4.5.1 To control the flow of the program, students used repeat-blocks and checked conditions to stop the loop.

In many quiz programs, students used repeat block to give users multiple chances to answer questions. In this case, students could give unlimited trials or set amount of trials. Regarding unlimited trials, students simply used a "repeat until" block with the specification of the correct answer. To set times of trials, students needed to use variables to count the trials. Q13 defined a variable, Trials, to trace times of trials and stopped the loop once the value reached three as Figure3-a describes. A student also stopped the loop using a "break" method. For example, Q11 gave three chances to answer a question. It used the "stop this script" block to stop the loop when a user answered correctly (see Figure3-b).

4.5.2 Students used forever-block to wait until a particular event occurred.

In maze game programs, students used the forever-block to make the code be responsive to a certain event such as "touching" a color or a sprite. For example, G03 set the forever-block in conjunction of "when flag clicked" block so the event specified within the forever-block could be detected continuously (see Figure3-c). This usage of forever-blocks should be guided to students because it is the unique way of Scratch utilizing loop for that purpose.

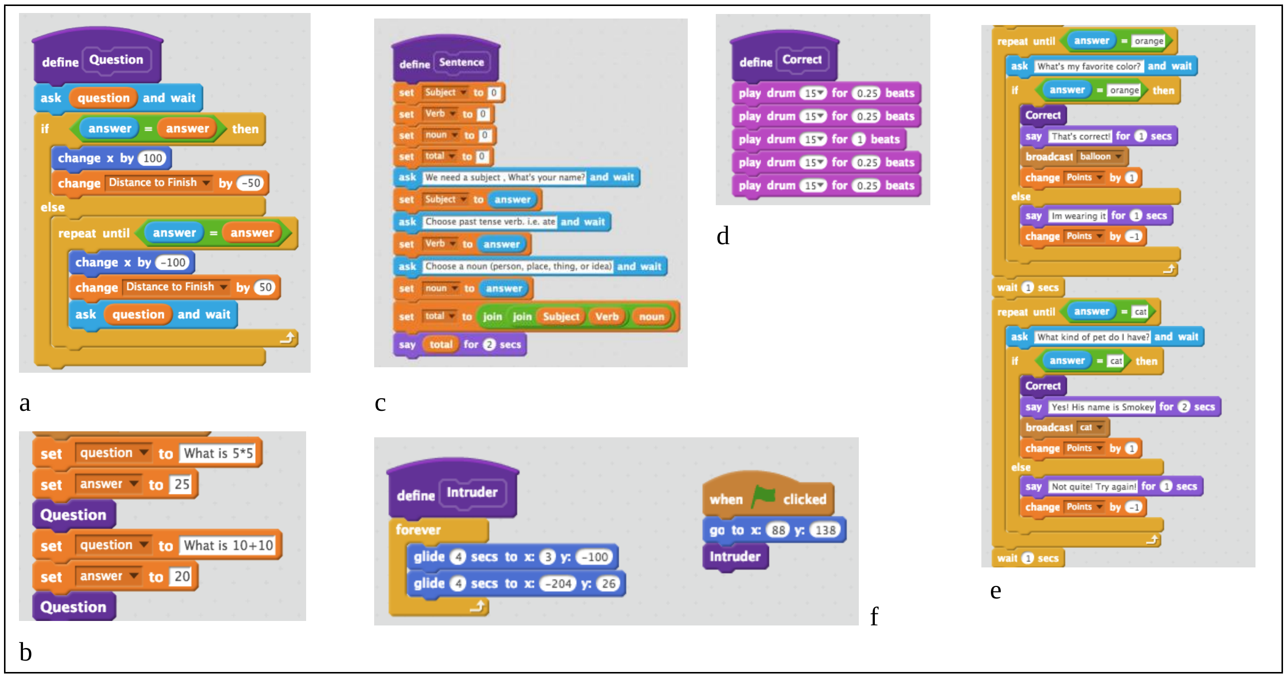
In common program language, such as JavaScript, an event handler covers this method. Scratch provides major event handlers like mouse clicked or keyboard inputs as a form of built-in functions. For example, Scratch has "When right-arrow-key pressed" which was identical to "Forever if 'key right arrow pressed' then" (compare Figure3-d and Figure3-e). However, the usage of forever-block for the event handling, called user-defined event handlers, requires a deeper understanding of event handling for the students to apply it to their program code than that of using built-in functions (Lee, 2010).

4.5.3 Usages of loops

While most game programs used the forever-blocks (8 out of 10), most quiz programs used the repeat <times> or repeat until <conditions> (11 out of 13). One quiz program did not use the repeat-block. In general, this result suggests that students selected the types of loops according to the purpose of the program.

4.6 Abstractions

Scratch allows students to create their own blocks by defining a set of blocks, so-called user-defined blocks. It is noteworthy there are efficient as well as inefficient ways to create the user-defined blocks. To decide the efficiency we reviewed how many times a user-defined block was reused and whether it used an argument. The ability to utilize the user-defined block is related to the competence of abstraction and problem decomposition.



a

```
define Question
ask (question) and wait
if <(answer) = (answer)> then
  change x by 100
  change Distance to Finish by -50
else
  repeat until <(answer) = (answer)>
    change x by -100
    change Distance to Finish by 50
    ask (question) and wait
```

b

```
set question to What is 5*5
set answer to 25
Question
set question to What is 10+10
set answer to 20
Question
```

c

```
define Sentence
set Subject to 0
set Verb to 0
set noun to 0
set total to 0
ask We need a subject , What's your name? and wait
set Subject to (answer)
ask Choose past tense verb. i.e. ate and wait
set Verb to (answer)
ask Choose a noun (person, place, thing, or idea) and wait
set noun to (answer)
set total to (join (join (Subject) (Verb)) (noun))
say (total) for 2 secs
```

d

```
define Correct
play drum 15 for 0.25 beats
play drum 15 for 0.25 beats
play drum 15 for 1 beats
play drum 15 for 0.25 beats
play drum 15 for 0.25 beats
```

e

```
repeat until <(answer) = orange>
  ask What's my favorite color? and wait
  if <(answer) = orange> then
    Correct
    say That's correct! for 1 secs
    broadcast balloon
    change Points by 1
  else
    say Im wearing it for 1 secs
    change Points by -1
wait 1 secs
repeat until <(answer) = cat>
  ask What kind of pet do I have? and wait
  if <(answer) = cat> then
    Correct
    say Yes! His name is Smokey for 2 secs
    broadcast cat
    change Points by 1
  else
    say Not quite! Try again! for 1 secs
    change Points by -1
wait 1 secs
```

f

```
define Intruder
forever
  glide 4 secs to x: 3 y: -100
  glide 4 secs to x: -204 y: 26

when flag clicked
go to x: 88 y: 138
Intruder
```

Figure 4. Codes representing the concepts of abstractions

4.6.1 Most efficient user-defined blocks

Q05 demonstrated the most efficient way to create blocks and use them in the code. The blocks were defined to generate quizzes and respond to user inputs accordingly. Q05 analyzed the pattern of quizzes (asking a question, receiving a user input, deciding whether it is correct or not, updating a score) and defined the blocks according to the pattern. It was impressive that Q05 used variables (arguments) to generate multiple questions by updating the values of the variables, which reduce the code complexity as well as the length (see Figure4-a and Figure4-b).

Q09 also defined a modifiable user-defined block. In this case, Q09 used variables to save user inputs and display an outcome accordingly. In this way, Q09 allowed the block to be reusable according to user inputs (see Figure4-c).

As Q09 did, other quiz programs also used variables to update specific values in a user-defined block. The purpose of the user-defined block was to create a quiz based on predefined question and correct answer. For this purpose, one used variables out of the block while the other used arguments within the user-defined block.

4.6.2 Efficient but limited user-defined blocks

Q03 defined a six-line code as a block that played drum sounds (see Figure4-d). A5 used the block whenever a user answered a question correctly. So, it reduced the complexity of the code and made it more manageable. However, close examination of Q03 codes revealed that asking a question and responding to it were the repeating construct that could be defined as a block (see Figure4-e). If two blocks, asking quiz and playing a drum, had been defined, the code would be much simpler and efficient.

Q04 and G09 also demonstrated the same issue in defining a block. In contrast to the efficiently programmed programs, they repeatedly used sets of blocks sharing same structure that could be replaced with user-defined blocks. This suggests that students should be trained to analyze patterns of codes and define a block to make the code simple and reusable.

Q11 used the user-defined block to control the flow of the program. After running a block, it called the next block at the end. Using this way, each block called the next block until the required flow ended. This way allowed Q11 to make the code "segmented" and easy to maintain. However, it required lots of inefficiently repeated codes.

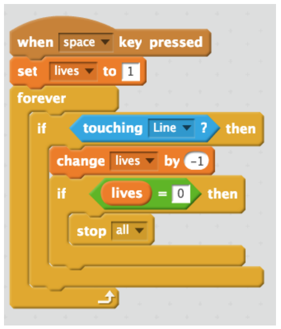
4.6.3 Inefficient user-defined blocks

G03 demonstrated the most common inefficient way to create the user-defined block. G03 defined a simple loop (moving sprite in a certain route) as a block and used it once for the sprite. This method failed to use the benefits of the user-defined block and even increased the complexity of the code unnecessarily (see Figure4-f). A total number of 13 programs demonstrated the same inefficient way other than to create the user-defined block: defining a simple code as a block and use the block once.

A few game programs, such as G03, G06, G07, and G08, duplicated the same structure of a block to other sprites. Although they can apply this concept in a common programming language by defining a function, it is not allowed in Scratch. This limitation may need to be explained to the students, so they do not overgeneralize that the scope of the user-defined block (function in other program languages) is limited to a certain sprite (file).

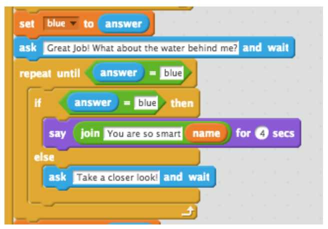
4.7 Variables

Using variables is one of the most crucial skills in computing. A variable is a tricky concept to grab (Kohn, 2017a, 2017b; Samurcay, 1989; Shi et al., 2018). Overall students defined variables and used them appropriately according to the purpose of the program. G01, for example, defined a variable named "Gift Money Left" and updated its value whenever a user used the money for a gift. Many game programs defined a score to update the points a user got or lost during the games. The variable had been used to decide whether a user succeeded or failed. Programs using variables to make a decision as well as to update values demonstrated competence regarding data representation.



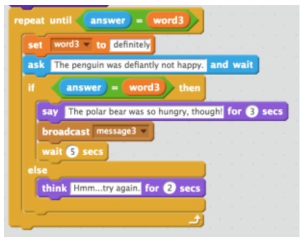
```
when space key pressed
set lives to 1
forever
    if touching Line ? then
        change lives by -1
        if lives = 0 then
            stop all
```

a



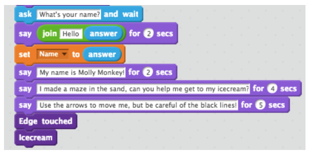
```
set blue to answer
ask Great Job! What about the water behind me? and wait
repeat until answer = blue
    if answer = blue then
        say join You are so smart name for 4 secs
    else
        ask Take a closer look! and wait
```

b



```
repeat until answer = word3
    set word3 to definitely
    ask The penguin was defiantly not happy. and wait
    if answer = word3 then
        say The polar bear was so hungry, though! for 3 secs
        broadcast message3
        wait 5 secs
    else
        think Hmm...try again. for 2 secs
```

c



```
ask What's your name? and wait
say join Hello answer for 2 secs
set Name to answer
say My name is Molly Monkey! for 2 secs
say I made a maze in the sand, can you help me get to my icecream? for 4 secs
say Use the arrows to move me, but be careful of the black lines! for 5 secs
Edge touched
Icecream
```

d

Figure 5. Codes representing the concepts of variables

4.7.1 Efficient use of variables

Half of the programs demonstrated competence to use variables effectively to set game times and trials, update scores, receive user inputs, and identify the highest scores (e.g., Figure5-a). They often integrated variables into repeat-blocks or if-blocks and controlled the flows of the program according to the values of variables.

G10 used a variable to calculate a sum of numbers, which is one of the most common usages of variables in arithmetic formulas. Q02 used a variable to present random choices. The variable was assigned a random number and was integrated into an if-block, so a specific instruction ran randomly. Some programs also used join-blocks to combine multiple values of different variables into one.

As mentioned in the abstractions (section 4.6), students used variables to specify certain values used in user-defined blocks. By integrating variables and user-defined blocks, students created reusable blocks that could be tailored by variables, which reduced the volume of code. Many quiz generating programs used this method.

4.7.2 Inefficient use of variables

Students sometimes never used defined variables. G03, G05, and Q03, for example, defined a variable that counted a success of a trial. Although they updated its value whenever a user completed a mission or answered a question correctly, no other code used the variable. So, the values of variables were never used for the programs meaningfully. For another example, Q04 defined "Difficulty" to identify the level of the quiz. Although it updated its values: HARD, MEDIUM, and EASY accordingly, the variable never cooperated with the other codes.

A few students were confused with the value and name of variables. Q01 did not seem to figure out the difference between the value of a variable and its name. As Figure5-b illustrates, the 'blue' variable saved 'answer' in it. (Please note the code was not correct because the first "answer" kept the previous user input rather than the current one as the program intended.) And the following repeat- and if-blocks used the static text "blue" to check whether the answer was correct. In this case, the correct answer was "blue" which was the same as the variable's name. To fix the code, Q01 might define "correctAnswer," set its value to blue, and used it in the following blocks. In the similar context, Q12 used the variable correctly by using the variable after assigning a correct answer (see Figure5-c).

In a few cases, students assigned a value to a variable after it was called, which suggests the flow of program was not appropriately considered. G04 defined "Name" to save a user's name. As figure5-d describes, the user input was used to make a greeting, "Hello answer." The "Name" variable received the input after the process, but G04 never used the variable in any other codes.

5. Discussion

The current study presents the evaluation results of two different approaches. Dr. Scratch provided a quick assessment for the Scratch programs. The comparisons between the quantitative complexity of the programs and the scores Dr. Scratch provided revealed strong positive correlations. In general, Dr. Scratch evaluations suggested that the students demonstrated the *developing* level rather than the *proficient* level. The assessments also revealed that there were considerable variations regarding the quality of the Scratch programs, especially in parallelism and synchronization. These two criteria emphasize the logical organization of events to make things happen in the order as designed. There are efficient and less efficient ways to achieve the task on Scratch. Considering that Scratch allows multiple ways to program a particular task, students should be guided to review various methods and evaluate their effectiveness while developing programs. It seems to be beneficial if students utilize Dr. Scratch during the programming processes to see the evaluations for a formative evaluation purpose. Although it does not suggest a solution for the program, students will be motivated to consider other ways to program.

The manual analysis of the Scratch programs revealed students' computational concepts in the context of their programming goals and tasks. Considering the short period of training sessions, students demonstrated a sufficient understanding of the main concepts and computing competency by applying the concepts into their programs. However, several issues needed to be improved. First, students should be able to eliminate unnecessary codes. The common mistakes observed in the study were related to redundant codes. For example, if there is one correct answer and three options are wrong, one will use if-else block and define else as all the wrong answers. In this case, one does not need to specify the conditions of three wrong answers if tailored feedback will not be given to each option. The analysis revealed students often added redundant codes that may increase the complexity and the chance for errors.

Second, students need to understand the program's specific characteristics. Many teachers utilize Scratch for introductory computing education because Scratch's features facilitate conceptual understandings by adopting a visual programming environment (Maloney et al., 2010). There are, however, some concepts and usages that students need to understand. The use of forever-blocks for an event handler is a unique feature of Scratch. Students often forget to include the infinite loop to make event-handling codes active (Lee, 2010). For this, teachers need to explain the function of loops in that particular context. As discussed, Scratch also provides built-in event-handlers, like detecting a keystroke from a user. Thus, it is recommended to provide a clear demonstration of how the built-in event-handlers are equivalent to other ways of making the function, so students can evaluate various methods of computing (Lee, 2010).

Repeat-until-blocks, for example, also need to be compared with another programming syntax, such as a while loop. In Scratch, the repeat-until-block executes its code until the condition becomes true. In other words, the repeat-block runs codes when the specified condition is "false" and stops the execution once the condition becomes "true." In contrast, while loops run their code when the specified condition is "true" and stop their execution once it becomes "false." To utilize Scratch for the transition to common text-based program languages, teachers need to emphasize the unique features of Scratch.

Third, teachers need to emphasize the ability to decompose problems and design solutions. The ways user-defined blocks were utilized revealed students' competence of problem decomposition. Those who figured out the pattern of codes could define new blocks efficiently. They even used variables and arguments in conjunction with the blocks so they could reuse them with different contexts. However, students who did not break down their problems or could not identify the patterns of solution processes developed inefficient and repeating codes. Program design needs to be taught with computational thinking.

Although the current study contributes to the literature by presenting multiple approaches to the evaluation of computational concepts, there are some limitations to be considered. As discussed, the two assessment methods, Dr. Scratch and manual evaluations, have unique strengths and weaknesses. The comparisons of two methods will validate the evaluation framework of computational concepts. Due to the small number of participants and the mismatch of the evaluation frameworks, the current study could not carry out the comparisons. Considering the potential contribution of mapping multiple evaluations of programming competency, further research exploring and validating various evaluation frameworks is highly recommended.

References

Aivaloglou, E., & Hermans, F. (2016). *How Kids Code and How We Know: An Exploratory Study on the Scratch Repository*. Paper presented at the Proceedings of the 2016 ACM Conference on International Computing Education Research, Melbourne, VIC, Australia.

Arraki, K., Blair, K., Bürgert, T., Greenling, J., Haebe, J., Lee, G., . . . Hug, S. (2014). *DISSECT: An experiment in infusing computational thinking in K-12 science curricula.* Paper presented at the 2014 IEEE Frontiers in Education Conference (FIE).

Arzarello, F., Chiappini, G. P., Lemut, E., Malara, N., & Pellerey, M. (1993). Learning Programming as a Cognitive Apprenticeship Through Conflicts. In E. Lemut, B. du Boulay, & G. Dettori (Eds.), *Cognitive Models and Intelligent Environments for Learning Programming* (pp. 284-298). Berlin, Heidelberg: Springer.

Bau, D., Gray, J., Kelleher, C., Sheldon, J., & Turbak, F. (2017). Learnable programming: blocks and beyond. *Communications of the ACM, 60*(6), 72-80. doi:10.1145/3015455

Buffum, P. S., Lobene, E. V., Frankosky, M. H., Boyer, K. E., Wiebe, E. N., & Lester, J. C. (2015). *A Practical Guide to Developing and Validating Computer Science Knowledge Assessments with Application to Middle School*. Paper presented at the Proceedings of the 46th ACM Technical Symposium on Computer Science Education, Kansas City, Missouri, USA.

Buitrago Flórez, F., Casallas, R., Hernández, M., Reyes, A., Restrepo, S., & Danies, G. (2017). Changing a Generation's Way of Thinking: Teaching Computational Thinking Through Programming. *Review of Educational Research, 87*(4), 834-860. doi:10.3102/0034654317710096

Buss, A., & Gamboa, R. (2017). Teacher Transformations in Developing Computational Thinking: Gaming and Robotics Use in After-School Settings. In P. J. Rich & C. B. Hodges (Eds.), *Emerging Research, Practice, and Policy on Computational Thinking* (pp. 189-203). Cham: Springer International Publishing.

Chao, P.-Y. (2016). Exploring students' computational practice, design and performance of problem-solving through a visual programming environment. *Computers & Education, 95*, 202-215. doi:10.1016/j.compedu.2016.01.010

Cooper, S., & Cunningham, S. (2010). Teaching computer science in context. *ACM Inroads, 1*(1), 5-8. doi:10.1145/1721933.1721934

Google, & Gallup. (2015). *Searching for Computer Science: Access and Barriers in U.S. K-12 Education*. Retrieved from https://goo.gl/oX311J

Google, & Gallup. (2017). *Encouraging students toward computer science learning. Results from the 2015-2016 Google-Gallup study of computer science in U.S. K-12 schools*. Retrieved from https://goo.gl/iM5g3A

Grover, S., & Basu, S. (2017). *Measuring Student Learning in Introductory Block-Based Programming: Examining Misconceptions of Loops, Variables, and Boolean Logic*. Paper presented at the Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education, Seattle, Washington, USA.

Grover, S., & Pea, R. (2013). Computational Thinking in K–12: A Review of the State of the Field. *Educational Researcher, 42*(1), 38-43. doi:10.3102/0013189x12463051

Grover, S., Pea, R., & Cooper, S. (2015). Designing for deeper learning in a blended computer science course for middle school students. *Computer Science Education, 25*(2), 199-237. doi:10.1080/08993408.2015.1033142

Kalelioğlu, F., & Gülbahar, Y. (2014). The effects of teaching programming via Scratch on problem solving skills: A discussion from learners' perspective. *Informatics in Education-An International Journal*(Vol13_1), 33-50.

Kohn, T. (2017a). *Variable Evaluation: an Exploration of Novice Programmers' Understanding and Common Misconceptions*. Paper presented at the Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education, Seattle, Washington, USA.

Kohn, T. (2017b). *Variable Evaluation: an Exploration of Novice Programmers' Understanding and Common Misconceptions*. Paper presented at the ACM SIGCSE Technical Symposium on Computer Science Education, Seattle, Washington, USA.

Kwon, K. (2017). Novice programmer's misconception of programming reflected on problem-solving plans. *International Journal of Computer Science Education in Schools, 1*(4), 14-24. doi:10.21585/ijcses.v1i4.19

Lahtinen, E., Ala-Mutka, K., & Järvinen, H.-M. (2005). A study of the difficulties of novice programmers. *ACM SIGCSE Bulletin, 37*(3), 14-18. doi:10.1145/1151954.1067453

Lee, Y. (2010). Developing computer programming concepts and skills via technology-enriched language-art projects: A case study. *Journal of Educational Multimedia and Hypermedia, 19*(3), 307-326.

Liu, C.-C., Cheng, Y.-B., & Huang, C.-W. (2011). The effect of simulation games on the learning of computational problem solving. *Computers & Education, 57*(3), 1907-1918. doi:10.1016/j.compedu.2011.04.002

Maloney, J., Resnick, M., Rusk, N., Silverman, B., & Eastmond, E. (2010). The Scratch Programming Language and Environment. *ACM Transactions on Computing Education, 10*(4), 1-15. doi:10.1145/1868358.1868363

Meerbaum-Salant, O., Armoni, M., & Ben-Ari, M. (2013). Learning computer science concepts with Scratch. *Computer Science Education, 23*(3), 239-264. doi:10.1080/08993408.2013.832022

Moreno, J., & Robles, G. (2014, 22-25 Oct. 2014). *Automatic detection of bad programming habits in scratch: A preliminary study*. Paper presented at the 2014 IEEE Frontiers in Education Conference (FIE) Proceedings.

Moreno-León, J., Robles, G., & Román-González, M. (2015). Dr. Scratch: Automatic analysis of scratch projects to assess and foster computational thinking. *RED. Revista de Educación a Distancia*(46), 1-23.

Moreno-León, J., Robles, G., & Román-González, M. (2016). Code to Learn: Where Does It Belong in the K-12 Curriculum? *Journal of Information Technology Education, 15*, 283-303.

Reding, T. E., Dorn, B., Grandgenett, N., Siy, H., Youn, J., Zhu, Q., & Engelmann, C. (2016). *Identification of the Emergent Leaders within a CSE Professional Development Program*. Paper presented at the the 11th Workshop in Primary and Secondary Computing Education, Münster, Germany.

Resnick, M., Maloney, J., Monroy-Hernandez, A., Rusk, N., Eastmond, E., Brennan, K., . . . Kafai, Y. (2009). Scratch: programming for all. *Communications of the ACM, 52*(11), 60-67. doi:10.1145/1592761.1592779

Sáez-López, J.-M., Román-González, M., & Vázquez-Cano, E. (2016). Visual programming languages integrated across the curriculum in elementary school: A two year case study using "Scratch" in five schools. *Computers & Education, 97*, 129-141. doi:10.1016/j.compedu.2016.03.003

Samurcay, R. (1989). The concept of variable in programming: Its meaning and use in problem-solving by novice programmers. In E. Soloway & J. C. Spohrer (Eds.), *Studying the novice programmer* (pp. 161-178). Hillsdale, NJ: Lawrence Erlbaum.

Shi, N., Cui, W., Zhang, P., & Sun, X. (2018). Evaluating the Effectiveness Roles of Variables in the Novice Programmers Learning. *Journal of Educational Computing Research, 56*(2), 181-201. doi:10.1177/0735633117707312

Simsek, A. (2011). The Relationship between Computer Anxiety and Computer Self-Efficacy. *Contemporary Educational Technology, 2*(3), 177-187.

Su, A. Y. S., Yang, S. J. H., Hwang, W., Huang, C. S. J., & Tern, M. (2014). Investigating the role of computer-supported annotation in problem-solving-based teaching: An empirical study of a Scratch programming pedagogy. *British Journal of Educational Technology*, *45*(4), 647-665. doi:10.1111/bjet.12058

Topalli, D., & Cagiltay, N. E. (2018). Improving programming skills in engineering education through problem-based game projects with Scratch. *Computers & Education*, *120*, 64-74. doi:10.1016/j.compedu.2018.01.011

Wing, J. M. (2006). Computational thinking. *Communications of the ACM*, *49*(3), 33-35. doi:10.1145/1118178.1118215

Winslow, L. E. (1996). Programming pedagogy - a psychological overview. *ACM SIGCSE Bulletin, 28*(3), 17-22. doi:10.1145/234867.234872

Yadav, A., Hong, H., & Stephenson, C. (2016). Computational Thinking for All: Pedagogical Approaches to Embedding 21st Century Problem Solving in K-12 Classrooms. *TechTrends*, *60*(6), 565-568. doi:10.1007/s11528-016-0087-7