IJCSES

Volume 2, Issue 3
August 2018
ISSN 2513-8359

International Journal of Computer Science Education In Schools

Editors

Dr. Filiz Kalelioglu

Yasemin Allsop

www.ijcses.org

International Journal of Computer Science Education in Schools

August 2018, Vol 2, No 3

DOI: https://doi.org/10.21585/ijcses.v2i3

Table of Contents

Teaching Computer Science Concepts through Robotics to Elementary School Children*

Mor Friebroon-Yesharim[1]
Mordechai Ben-Ari[1]

[1]Weizmann Institute of Science

DOI: 10.21585/ijcses.v2i3.30

Abstract

Studying computer science (CS) in elementary schools has gained become popular in recent years. However, students at such a young age encounter difficulties when first engaging with CS. Robotics has been proposed as a medium for teaching CS to young students, because it reifies concepts in a tangible object, and because of the excitement of working with robots. We asked: What CS concepts can elementary-school students learn from the participation in a robotics-based CS course? We used two theoretical frameworks to explain possible difficulties in learning: the Jourdain effect, and constructs vs. plans. A taxonomy of six levels was created to characterize levels of learning. The levels were measured using four questionnaires that were based on the taxonomy. In addition, field observations of the lessons were recorded. The population consisted of 118 second-grade students (ages 7-8). Lessons on CS concepts using Thymio educational robot and its graphical software development environment were taught during normal school hours, not in a voluntary extracurricular activity. The syllabus was based on existing learning materials that were adapted for the young age of the students. The analysis showed that the students were very engaged with the robotics activities. They did learn basic CS concepts, although they found it difficult to create and run their own programs. We concluded that the Jourdain effect was not demonstrated because the students understood concepts and constructs of CS; however, they were unable to plan and construct their own programs from the basic constructs.

Keywords: elementary school, computer science, Thymio robot, Braitenberg creatures

* Preliminary results of this research were presented at the International Conference on Robotics in Education, April 2017, Sofia, Bulgaria.

1 Introduction

1.1. Research goals and research framework

Studying computer science (CS) in elementary schools has gained become popular in recent years. The goal of teaching CS at a young age is primarily to increase self-efficacy and motivation when engaging with science and technology. However, students at such a young age face difficulties when first engaging with CS. One approach to overcome these difficulties is to use robotics activities, because they reify abstract CS concepts in a tangible object and because of the excitement of working with robots. The goal of our research is to distinguish between the performance of a task and the understanding of the concepts.

The phases of the research were:

1) The development of an age-appropriate syllabus partially based on existing learning materials for the educational robot.
2) A quantitative and qualitative assessment to determine if the use of a robot-based syllabus enables young students to understand CS concepts.
3) The development of taxonomy appropriate for specifying the levels of understanding of young students who learn CS.

Section 1 presents a review of existing literature. The methodology, including the new taxonomy, is described in Section 2. The findings are presented in Section 3, discussed in Section 4 and summarized in Section 5.

1.2. Review of existing literature

The literature review is divided into five sections: (a) learning CS concepts by elementary school students, (b) learning CS by robotics, (c) learning CS with robotics in elementary school, (d) the Jourdain effect, (e) near transfer.

1.2.1. Learning CS concepts by elementary school students

Papert was among the first to propose teaching programming to young children (Papert, 1980). He coined the term constructionism for learning by constructing artifacts such as computer programs. Research has shown that CS studies had some positive effects on cognitive development, thinking skills, problem-solving strategies, creativity, intrinsic motivation, or even social development specifically at young ages (Liao & Bright, 1991; Clements, 2002; Clements & Sarama, 2003). Duncan and Bell (2015) explored and analyzed CS curricula for elementary schools; they found that some countries have already incorporated formal studies as part of the curriculum, while others are limited to informal classes and clubs. Seiter and Foreman (2013) explored the development of computational thinking by elementary school students using Scratch. They found that basic proficiency of algorithmic thinking started in second grade. Clements and Sarama (1997) studied

learning with LOGO and concluded that it can provide an evocative context for young children's explorations of mathematical ideas and CS concepts.

Duncan et al. (2014) raised the question: Should eight-years-old students learn to code? They proposed three parameters to establish effective learning: (1) the teachers must be confident and motivated; (2) the learning objectives should be realistic for the age of the students; (3) the development environment should be age-appropriate. They found that the age at which programming should be taught depends on many factors among them the software tools and learning aids, the context and the teachers' training. Armoni and Gal-Ezer (2014) and Duncan et al. (2014) claimed that the advantages of learning CS at such a young age include the capability: to learn quickly, to shape attitudes to programming, to support learning outside of just programming, and to prepare students for future endeavors in computing. The disadvantages of engaging with CS at a young age include the possibility that students will be less confident in their abilities regarding CS or will receive a negative impression of the subject. Furthermore, the students may study fewer hours in core subjects such as mathematics, science and language skills (Duncan & Bell, 2015). The limited time available and the lack of resources could cause problems in the allocation of school resources (Duncan et al., 2014).

1.2.2. Learning CS via robotics

Since the 1980s, both curricular courses and outreach programs on robotics have been developed. A pioneering successful tool for teaching CS with robotics was the environment Karel the Robot (Pattis, 1981). Anderson et al. (2011) claimed that robotic activities are very exciting for the students and that they reify the abstract behavior of algorithms and programs. Robotics provides hands-on experience with real-world problems and can also reduce the level of intimidation that students can encounter. Ben-Bassat-Levy and Ben-Ari (2015) showed that robotic activities can influence both the motivation and the self-efficacy of young students. They found that robotics encourages positive intentions to choose STEM (science, technology, engineering, mathematics) subjects in high school. Markham and King (2010) investigated attitudes and motivation among CS1 students, they found that students who studied with robots had more positive experiences than those who studied without robots. Kaloti-Hallak et al. (2015) investigated young students who participated in the FIRST® LEGO® League competitions. Their research showed that students demonstrated meaningful learning in computer science and engineering, and that most of the students demonstrated high positive attitudes and motivation for learning robotics. Kay (2011) showed that CS learning by high-school and undergraduate students really improve when robots are used.

1.2.3. Learning CS with robotics in elementary school

Barker and Ansorge (2007) taught 9–11 year-old students and found that the LEGO Mindstorms® robotics kit was effective for teaching STEM concepts. Magnenat et al. (2014) ran a workshop for students aged 8–9 using the Thymio educational robot. They found that while students successfully

used trial and error when writing programs that controlled the robot, they only understood a subset of the CS concepts that appeared in their programs. Both these projects involved extracurricular activities. Several research projects addressed learning with robotics by young children. Martinez et al. (2015) taught robotics to students of a variety of age groups from 3 to 11. They showed that the older students were capable of understanding and applying CS concepts such as loops, parameters, conditions and sequencing, while preschool students understood fewer concepts. Sullivan and Bers (2016) implemented a robotics curriculum in preschool through second grade. They found that younger children were able to master basic concepts of robotics and programming, while older children were able to master more complex concepts. Bers et al. (2014) engaged 4–6 year-old children in robotics activities in order to guide age-appropriate curriculum development. Wyeth (2008) showed that children can learn simple programming concepts related to input and output, and the impact of logic on program behavior. Common to all these research projects is the use of robots specifically designed for young children, in particular, Bers' group used tangible programming (programming using physical blocks), which can no longer be used for older students.

1.2.4. The Jourdain effect

Guy Brousseau proposed the *theory of didactic situations* as a framework for investigating learning, in particular, mathematics (Brousseau, 1997). We were influenced by his discussion of the *Jourdain effect*, the conflation of the performance of a task with understanding the underlying concepts. Here is an example from the New Math curriculum:

> [A] Model of this group can be constructed using some transformations of the position of a cup of yogurt … As children were playing with the cups of yogurt, they were performing such transformations … from this, the 'structurally minded' observers were concluding that the children 'have constructed' the group of Klein. But what the children were actually doing had nothing to do with the identification of the group structure in their manipulations. … [T]hey would not have been able to identify the part of their activity called 'construction of the group of Klein' … and they would not be able to produce …, further examples of their activity, now sanctified by a scientific term (Sierpinska, 2003, no page numbers).

1.2.5. Near transfer

It is frequently claimed that learning certain subjects results in transfer of knowledge: a general improvement of cognitive and problem solving abilities. Such claims have recently been made for computational thinking (see the analysis by Tedre and Denning (2016)). Such claims have been repeatedly debunked (Guzdial, 2015) and we make no such claims for learning CS with robotics. However, a classic investigation by Gick and Holyoak (1980) showed that learning in one domain can aid in solving *analogous* problems in a different domain; this is called *near transfer*. While our research is not a comprehensive study of near transfer, we did investigate whether the students were

able to transfer their knowledge of robotics commands to a hypothetical command somewhat different from the actual commands.

2 Methodology

2.1. Rationale for the research

There were three aspects to the rationale for this research:

1) The robotics activities were taught in the non-voluntary, non-selective environment of a normal classroom during school hours with just one teacher and one assistant. In previous work (Ber et al, 2014), several research assistants were able to help individual groups of students. Finally, a general purpose educational robot was used, not one specifically developed for the project.
2) We developed a questionnaire based on a taxonomy of learning in order to attempt to distinguish performance from understanding (the Jourdain effect).
3) We wanted to demarcate what this specific age group was able to learn from concepts that were too difficult for them.

2.2. Research question

What CS concepts can elementary school students understand from participation in a robotics-based CS course?

2.3. Population

The research was carried out in four second-grade classrooms of a public school (ages 7–8 years). All the students in these classes participated during normal school hours, so we had no control over the actual ages, genders or abilities of the students. There were 118 students, 72 boys and 46 girls. The first author taught the lessons aided by a research assistant from our department. The class teachers were present, but we did not have time to train them in robotics and CS so they were not involved in teaching. However, they remained in the classrooms, primarily to deal with behavioral problems that occasionally arose.

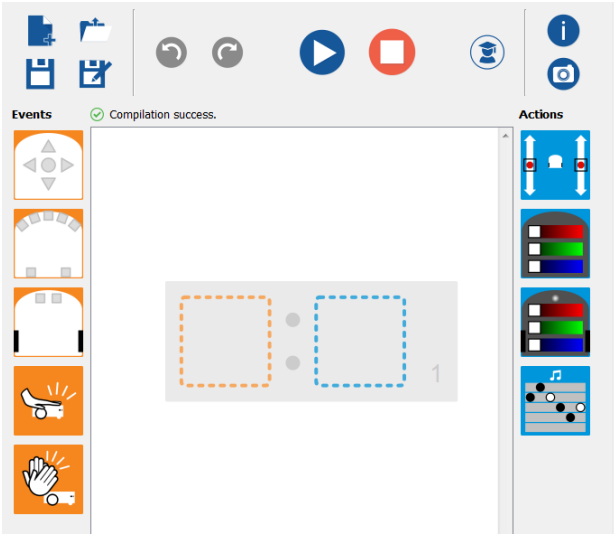
2.4. The robot and its software environment

The Thymio educational robot (Figure 1) is small, self-contained and very robust with differential drive, nine infrared proximity sensors, five touch-sensitive buttons, a 3-axis accelerometer, dozens of LEDs, a speaker and a microphone (https://www.thymio.org/en:thymio).

Figure 1. Thymio robot

The robot is programmed using the Visual Programming Language (VPL) environment (Figure 2) (Shin et al., 2014). Programs are constructed by drag-and-drop of graphical blocks. VPL supports one programming construct: event-action pairs. Event handling is a core CS concept, which has been proposed as the basis of teaching introductory programming (Bruce et al., 2006). In addition to the basic blocks shown in Figure 2, there is an advanced mode that supports additional blocks and advanced versions of basic blocks.

Figure 2. The VPL environment. The events are on the left, the actions are on the right and the central area is used for constructing programs



2.5. The robotics class

The course was taught for one hour a week for 21 weeks during normal school hours. Thirty students shared ten robots. Each lesson began with a short video that introduced a concept. Then the students received a worksheet.

The syllabus was based on existing learning materials for the Thymio (Ben-Ari, 2011; Magnenat et al., 2012), by selecting age-appropriate activities from this material. The first tasks were based on the predefined behaviors of the robot and were intended to familiarize the students with the events generated by the proximity sensors and the buttons, and with the actions of changing the colors of the LEDs. Then the students were introduced to the VPL graphical programming environment. Many of their tasks were to implement *Braitenberg creatures* (Braitenberg, 1984), which were developed during the Programmable Brick project (Hogg et al., 2000) that was the inspiration for LEGO Mindstorms®. Later, the students explored combinations of several actions per event and blocks from

the VPL's advanced mode: timer events and actions, accelerometer events and music actions.

In addition to the concepts explicitly expressed in the Thymio robot and VPL, the following CS concepts that appeared only implicitly were taught:

4) Concurrent execution of event-action pairs: for example, in the line-following program, the program simultaneously checks if the robot is leaving the left edge or the right edge of the line.
5) Parameters: setting the color of the LEDs and the power applied to each motor.
6) Writing an algorithm and implementing it in a program.

Table 1 presents the topics and activities that took place in each lesson.

Table 1. The content of the lessons

Lesson	Content
1	Pre-programmed behaviors.
2	Pre-programmed behaviors in groups.
3	The VPL user interface. Events (buttons, sensors) and actions (top and bottoms colored LEDs). First experience programming.
4	The VPL user interface. Programming five exercises.
5	First questionnaire (four questions).
6	The motor action. Programming the Braitenberg creatures.
7	First questionnaire (two questions). Second questionnaire (two questions). Checking the exercises from the previous class. Multiple actions for one event.
8	Second questionnaire (two questions). Braitenberg creatures with multiple actions per event.
9	Checking the Braitenberg creatures exercises.
10	The clap and tap sound events and the sound action.
11	Second questionnaire (two additional questions). Bottom sensors.
12	Programming with the bottom sensors.
13	Programming with the bottom sensors.
14	Programming with the bottom sensors (summary).
15	Third questionnaire (two questions). The advanced mode.
16	Accelerometers.
17	Fourth questionnaire (four questions).

	Timer blocks.
18	Fourth questionnaire (two questions). Exercises with the timer blocks.
19	Exercises with the timer blocks.
20	Group projects.
21	Group projects.

2.6. The taxonomy

Our method for investigating performance vs. understanding was based on a taxonomy of levels of learning. We developed a new taxonomy appropriate for the context of learning CS concepts.

2.6.1. Justification for developing a new taxonomy

There are several existing taxonomies of levels of learning: SOLO (Biggs & Collis, 1982), Bloom (Bloom et al., 1956), Lister et al. (2004) and the CS-specific taxonomy of Fuller et al. (2007). Meerbaum-Salant et al. (2013) combined the Bloom and SOLO taxonomies, producing a new scale with three categories (unistructural, multistructural and relational) each containing three sub-categories (understanding, applying, and creating). Magnenat et al. (2014) based their work on learning with the Thymio robot on the combined taxonomy. They investigated two age groups: 8-9 and 10-15 years old. They checked the levels of understanding using questionnaires. While most of the students of the older group achieved the level of unistructural understanding, the young age group found it hard to answer the questions because of their limited ability to read. The difficulties they encountered led us to develop a new taxonomy.

2.6.2. The new taxonomy

A student demonstrating the follow behaviors will be considered to have achieved a corresponding level of learning. In ascending order they are:

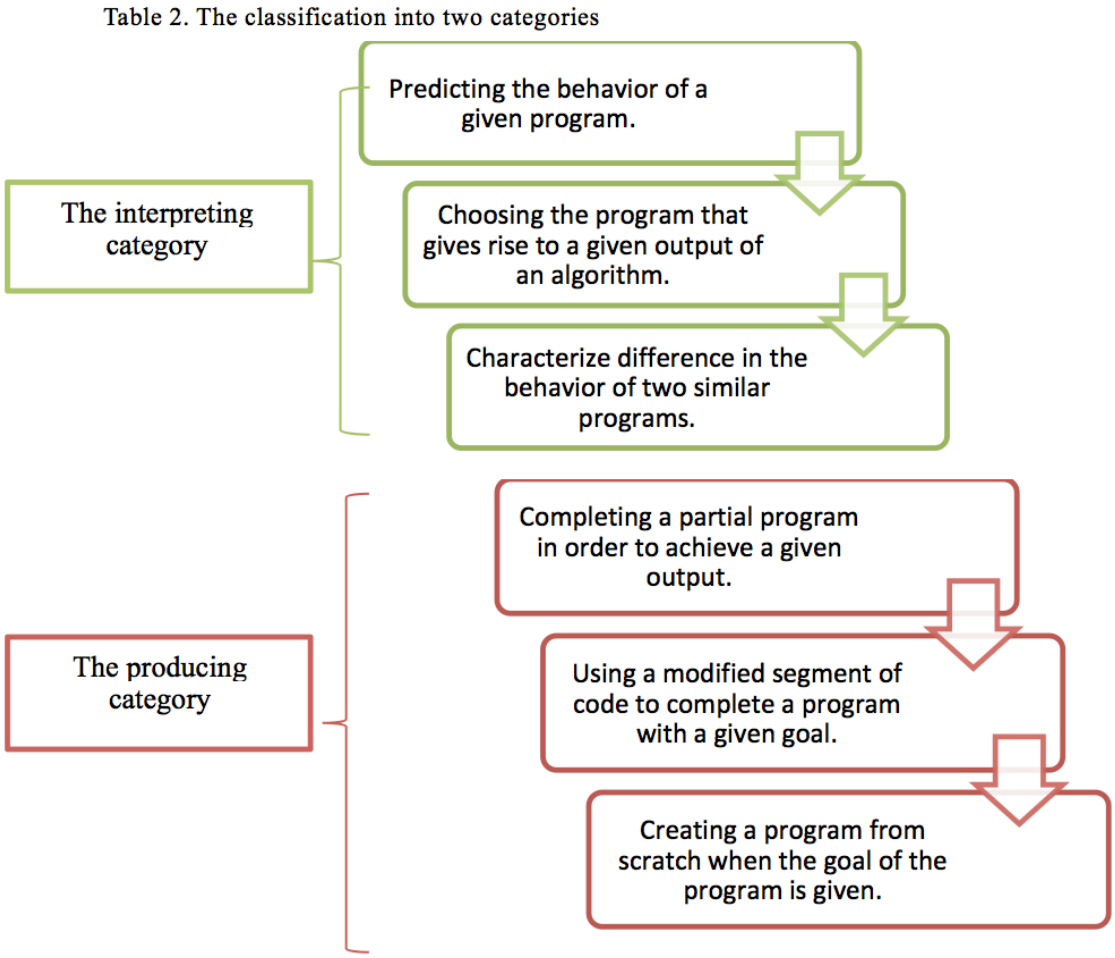
1) Predicting the behavior of a given program.
2) Choosing the program that gives rise to a given output of an algorithm.
3) Characterizing the difference in the behavior of two similar programs.
4) Completing a partial program in order to achieve a given output.
5) Using a modified programming construct to complete a program with a given goal.
6) Creating a program from scratch when the goal of the program is given.

The different types of behaviors can be classified into two categories (Table 2) based on Fuller et al. (2007):

1) Interpreting: to give or provide the meaning of; explain.
2) Producing: to bring into existence by intellectual or creative ability.

The interpreting category includes only behaviors for which the code is given, as oppose to the producing category whose behaviors require code completion. This enabled us to distinguish between students who can only read and understand programs but cannot necessarily write or complete one on their own.

Table 2. The classification into two categories



2.6.3. Secondary classification: The interpreting category

The ordering of the three behaviors in the interpreting category can be justified as follows.

Lister et al. (2004) claimed that the ability to predict the behavior of a given program while tracking and following its instructions is the lowest cognitive level required for a CS student. The second type of question differs from the first type in that the students are required to choose the right program from several slight different programs. Lister et al. (2004) claimed that students showed less success in this type of questions then in questions predicting the behavior of a program.

Differentiating between two programs is similar to comparing, but it adds the requirement to predict

the purpose of the programs. Differentiating also requires analytical skills to identify similarities and differences, which is a higher cognitive process than just predicting, according to the revised Bloom's taxonomy (Anderson et al. 2001). Therefore, this is the most challenging behavior in this category.

2.6.4. Secondary classification: The producing category

In this category, the student is asked to complete a program. Even if the questions are multiple-choice, this category requires the student to analyze the purpose of the program and to choose the most suitable completion. The student must understand the functionality of the missing parts, and then to understand how each possible completion will affect the program.

The second behavior and the question refer to it; the students are presented with modified block whose functionality is explained. They need to complete a given segment of code correctly using this modified block. The cognitive stages are very similar to the first behavior in this category, but, in addition, the student needs to show proficiency in the material he has already learned in order to understand the purpose of the modifications. We consider this process to be near transfer of knowledge and this type of behavior is harder than the first one.

The most challenging type of behavior in this category asks the students to create a program from scratch when the goal of the program is given. This requires the student to use all of the knowledge he has gained so far and to create a new artificat. Both the Bloom (Bloom et al., 1956) and SOLO (Biggs, & Collis, 1982) taxonomies rate this cognitive skill as the most challenging one.

2.6.5. Constructing questions according to the taxonomy

Magnenat el al. (2014) found that elementary school students had difficulty understanding long passages of text; this led them to use graphics and video clips in their questionnaires. We followed their lead in constructing our questionnaires. Graphics is particular appropriate in this context since the VPL programming environment uses graphical elements only with no text. Here is a description of the format of the questions associated with each level of the taxonomy:

7) Predicting the behavior of a given program: The students received a code segment and four photos of the robot and they had to choose the photo that demonstrated the behavior of the robot caused by the code segment.
8) Choosing the program that gives rise to a given output of an algorithm: The students received a short video and had to choose the code segment that gives rise to this behavior.
9) Characterizing differences in the behavior of two similar programs: The students watched two short videos displaying behaviors of the robot and they had to choose among four descriptions the one that describes the difference between the behaviors.
10) Completing a partial program in order to achieve a given output: The students were given a short segment of code and a goal that the program needs to achieve. The students had to complete the missing parts of the code in order for the program to achieve the goal.

11) Using a modified segment of code to complete a program with a given goal: The students were given a modified block whose meaning was explained in the question and they had to complete the code segment in order to implement the behavior demonstrated in a short video or explained in words.
12) Creating a program from scratch when the goal of the program is given: The students were given several event-action blocks and they had to choose and arrange the blocks needed in order to achieve a given goal.

2.7. Conjectures

We proposed three conjectures to explain why some students might achieve only lower levels of learning, together with criteria to accept or reject the conjectures:

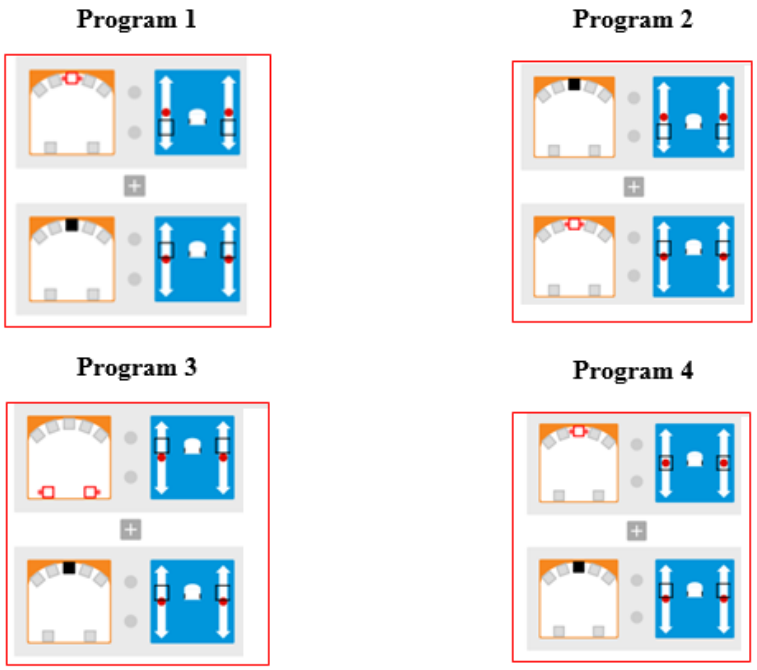
13) **They really don't understand** The simplest possibility is that young students don't understand most of the CS concepts that they are exposed to, and that the success reported in previous research has been misinterpreted. This conjecture will be supported if we find that less than 50% of the students succeed in answering questions even for the lowest levels of learning.
14) **Jourdain effect** The students will demonstrate the Jourdain effect if they successful answer the predicting and choosing types of questions, partial succeed in the answering difference questions, and are unable to answer any of the questions from the second category.
15) **Constructs vs. plans** Soloway and Spohrer (1986) suggest that there is a gap between the ability of novice programmers to understand individual constructs and their ability to plan and implement a functioning program. This conjecture will be supported if students are only able to answer questions from the interpreting category and the first question of the producing category. Students demonstrating the ability to implement plans will be able to answer questions from all the categories.

2.8. Research Instruments

Four questionnaires of six multiple-choice questions each were administrated during the regular lessons. They looked like the usual worksheets and the students willingly participated in solving the problems. After the first experience with the class assignments, it was decided that the maximum number of questions that these young students could deal with in one lesson is four, so the questionnaires were split over two or more lessons. The questionnaires can be found at https://goo.gl/peKBpp.

The topics asked about in the questionnaires are as follows:

1) The first questionnaire



CS concepts: simple event-action pairs.

New events and actions blocks: sensors, buttons, top and bottom colors.

2) The second questionnaire

CS concepts: advanced event-action pairs.

New events and actions blocks: motors.

3) The third questionnaire

CS concepts: multiple actions in one event-action pairs.

New events and actions blocks: tap and clap detection.

4) The fourth questionnaire

CS concepts: line following, concurrent execution.

New events and actions blocks: ground proximity sensors.

After each lesson, the first author recorded her observations and solicited impressions from the research assistant and the class teacher.

In order to fully investigate the ability of students to plan and implement a program, during the final lessons they were asked to come up with their own ideas for writing a program. The programs and observations of the programming process were recorded.

After the final lesson, the first author met with the teachers to discuss their impressions of the students' ability to learn CS and robotics.

2.8.2 Example questions from the questionnaires

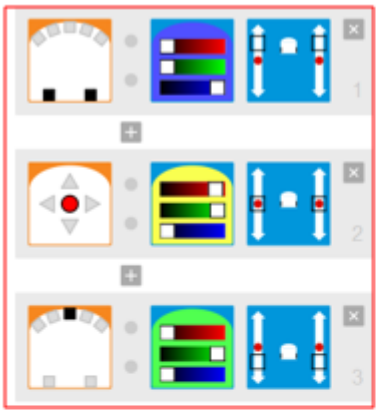
Question 2 from questionnaire 2

Watch the video: https://youtu.be/okoLamAY9ac. Which of the following programs causes the behavior of the robot that is shown in the video?

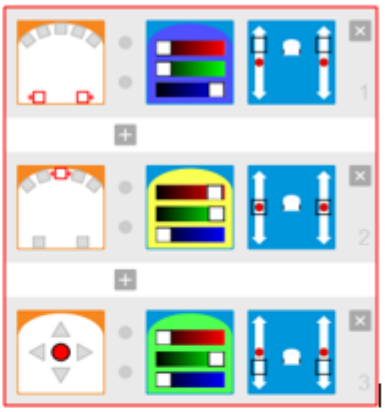
Question 2 from questionnaire 3

Look at the video: https://youtu.be/Vhc33fxR3co. Which of the following programs causes the behavior shown in the video?

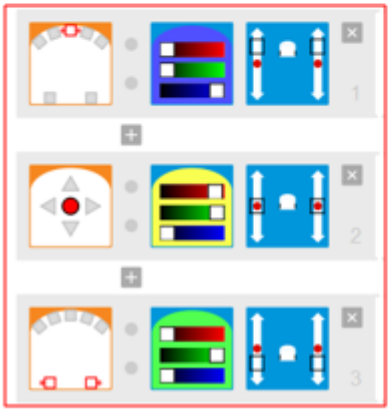
Program 1



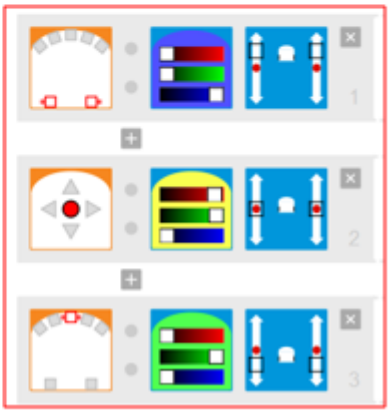
Program 2



Program 3

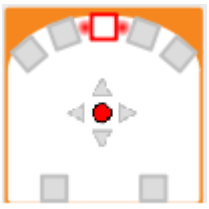


Program 4



Question 5 from questionnaire 1

We invented a new event: The center button is touched and at the same time an object is detected by the center front sensor. Here is the block for the new event:



Use the new event to construct a program that does the following:

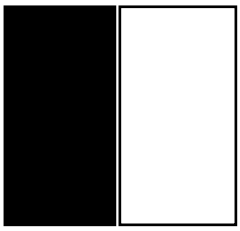
1) Detecting an object only by the front center sensor causes the top light to display blue.

2) Touching only the center button causes the top light to display yellow.
3) Touching the center button and at the same time detecting an object by the front center sensor causes the top light to display green.

Choose the correct program:

Question 5 from questionnaire 4

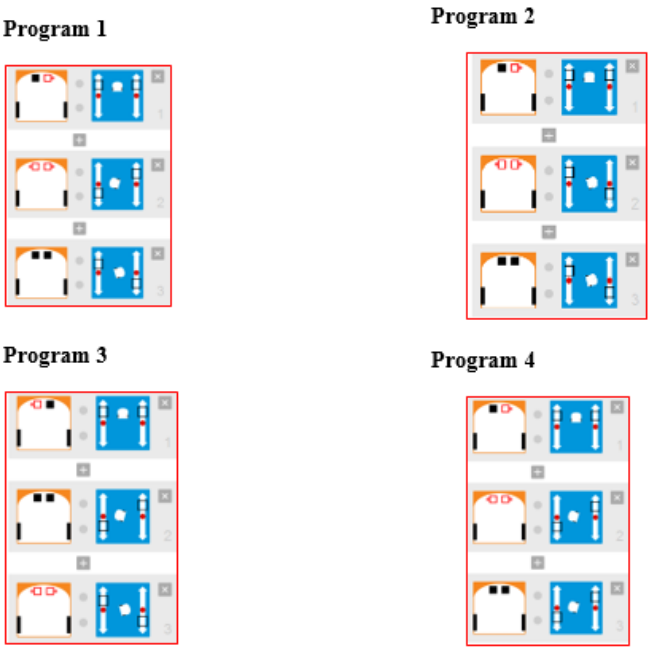
We want a program that causes Thymio to follow the edge between a white area and a black area:



Here is a description of the program:

1) If Thymio detects white under the right bottom sensor and it detects black under the left bottom sensor, then Thymio moves forwards.
2) If Thymio detects white under both bottom sensors, then Thymio turns left.
3) If Thymio detects black under both bottom sensors, then Thymio turns right

Choose the correct program:



2.9 Data Analysis

The findings from the observations will be integrated with a description of the syllabus, in order to show how the students reacted to each individual topic. The analysis was influenced by Glaser and Strauss (1967) and was done as follows:

1) Collate the observations of the four classes with the same lesson plan.
2) Identify the important events, and the similarities and differences among the four classes.
3) Unify similar events and important concepts.
4) Link the different categories in order to understand the students' capabilities.

The discussion of the focus group with the teachers was recorded and transcribed. The analysis of this focus group was identical to the analysis of the observations of the students.

The project lessons were analyzed in order to find the CS concepts that the students used and to discover the level of understanding that the students achieved.

The questionnaires were analyzed quantitatively using Pearson's chi-squared test, which is used to determine whether there is a significant difference between the expected frequencies and the observed frequencies in one or more categories. Since each question was asked in all of the four classes, and every question had four options, we found the chi-squared test most appropriate for the research needs. The chi-squared test was used to determine whether there was a significant difference between the expected frequencies and the observed frequencies in one or more of the four classes. For each questionnaire, a table is given which indicates the uniformity of the success rates of the four classes. The first column is the question number and the second column is the success rates.

For every question of the questionnaires, the null hypothesis was that there were differences among the four classes; the alternate hypothesis was that there were no differences in the success rates. Questions that did not negate the null hypothesis were marked with a gray background, and the

success percentages for those questions will be presented separately for each of the four classes.

When the percentage of the students who answered a question correctly passed 50% were marked with bold face, we took that as evidence that the level of the taxonomy corresponding to that question was achieved by the majority of the students.

In the presentation of the findings, we will emphasize the questions that didn't negate the expected frequencies and provide explanations in cases of discrepancy.

2.9.1 Reliability

Since the students had no previous background in CS, there was no reason to administer a pre-test. Colleagues were asked to judge that the curriculum and research methods involve well-known CS concepts and are age-appropriate. To check the reliability of the knowledge questionnaires, both colleagues and the teachers examined the questionnaires and expressed their opinions regarding their difficulty.

3 Findings

3.1. Initial experience with the robot

The first two lessons with the robot were designed to give the students an easy start, so they were conducted without a computer. During the lessons the students worked in groups. Each group had to perform several activities using the robot's pre-programmed behaviors (https://www.thymio.org/en:thymiostarting). In the second lesson, the students performed tasks with the help of a discovery kit (https://www.thymio.org/en:thymiodiscoverykitotter).

The students were excited; some girls said that the robot was "cute," while the boys described it as "cool" and indicated that Thymio is "fun." During the performance of the tasks, the students followed the instructions while correcting each other. When they were given open questions, they kept asking "what to write?" It seemed as if they wanted to be correct at every single question.

During class, the students managed to manipulate the robot and to switch between the robot's pre-programmed modes, but they found it hard to describe the differences between the modes. However, they were able to describe the different behaviors, as shown by their terminology: "it made a sound when I put both of my hands," "Thymio drove forward in the direction of the doll!" The students were not able to memorize the different modes, but they knew how to describe the modes during the activities.

3.2. Initial experience with programming

During the third and fourth lessons the students started to program. They were introduced to the VPL environment and used it to learn how to write programs.

The first worksheet with VPL included the following events: pressing the buttons, detecting objects with the front and back sensors, and the following actions: turn on the top and the bottom LEDs. The students were thrilled by their first experience writing programs, they were highly motivated to

succeed and readily proceeded from one exercise to the next. The students easily implemented the three tasks with a single event-action pair in this lesson. All the students seemed engaged with the class activities and the worksheet, asking questions and exploring the capabilities of the robot and VPL.

The students frequently asked "where to put it?" with regard to an event or an action; sometimes they used the buttons event without indicating which button, so they tended to ask many questions of the form "why isn't it working?" or "what am I doing wrong?" All the students wanted to be the ones with hands on the robot or the computer.

During the fourth lesson, the students were introduced to exercises that included more than one event-action pair. Initially, the students had difficulty understanding that they should be placed one after another on the screen. Although this was explained to the students, they found it unnatural, and asked "where to put this event?" or they just started a new program for each event-action pair. After sufficient practice they felt more confident with the concept of a program containing multiple event-action pairs.

3.3. The first questionnaire

The success rates of the first questionnaire (administered during lessons 5 and 7) are shown in Table 3. The first questionnaire investigated whether the students could associate an event with an action.

Table 3. First questionnaire, % of students who gave correct answers for each question

Question	Success Rate
1	**82**
2	**87**
3	**69**
4	**93**
5	**74**
6	**51**

For question 1, the α obtained from the chi-squared test was 0.0039; this result is greater than the likelihood ratio chi-squared, which is 0.0009, therefore the null hypothesis (that there were differences among the four classes as explained in section 2.9) cannot be rejected. Question 1 asked about the behavior of a given program. Although different success rates were obtained for the three classes, the success rates of all classes were greater than 50% (Table 4).

Table 4. The success rates of questionnaire number 1, question number 1 in the different classes

	Class A	Class C	Class D
Success Rate	**100**	**85**	**64**

From Table 3 we see that the students answered questions at levels 1 to 5 quite easily: (1) they could understand what a given program does, (2) match a program to an output, (3) characterize the difference in the behavior of two similar programs, (4) complete a partial program, (5) use a modified programming construct. However, for question 6, creating a program from scratch, their success rate was relatively low, but still reached 50%.

3.4. Programming the Braitenberg creatures

During the sixth lesson the students explored the motor block on their own. They copied programs given in the worksheet into the VPL environment, ran the programs and explained the behavior of each program: moving forwards, moving backwards and turning right and left. After understanding the motor block, the students wrote programs to implement the Braitenberg creatures. During the activities with the creatures, it was observed that these exercises helped the students "bond" with the robot and gave it some human qualities. The students began to refer to the robot as a living: "Thymio didn't do what I asked him!", "I miss Thymio," "I love Thymio."

3.5. The second questionnaire

Questionnaire 2 (administered during lessons 7, 8 and 11) investigated if the students can associate an event with an action, but this time they had more blocks at their disposal, which made the programs more complex. The success rates for the second questionnaire are shown in Table 5.

Table 5. Second questionnaire, % of students who gave correct answers for each question

Question no.	Success Rates
1	**52**
2	46
3	49
4	**76**
5	**70**
6	46

The first two questions of the questionnaires were given to the students just after their initial experience with the new motor block, while the next four questions were given after they had much more practice and experience with the robots.

The high success rates for questions four and five compared with those of questions one and two can be explained by the additional practice that the students had, which enabled them to answer questions at the higher levels. The students encountered difficulties answering question 3: it was hard for them to identify the difference between a pair videos showing behaviors of the robot. Nevertheless, in three of the four classes, a majority of the students successfully answered the question. The success rates for

question number 6 were low: the students found it difficult to write a program from scratch given the desired behavior of the robot.

The heterogeneity between the four classes was more significant in this questionnaire. For questions 3, 5 and 6, the α obtained from the chi-squared test were greater than the likelihood ratio chi-squared, so the null hypothesis could not be rejected.

Table 6 shows the α obtained from the chi-squared test and the likelihood ratio chi-squared for each of the questions.

Table 6. $\boldsymbol{\alpha}$ from the chi-squared test and the likelihood ratio chi-squared for each of the questions in questionnaire 2

Question	The α result obtained	The likelihood ratio chi-squared
1	0.1089*	0.05
2	0.0890	0.0862
3	0.0133	0.0105
4	0.5188*	0.5061
5	0.0506	0.05
6	0.0312*	0.05

*Because these three questions contained more than one section, the α obtained is the average of the differences between the four classes and actually represents the Generalized Linear Model (GLM) test.

It can be seen that the differences between the α obtained and the likelihood ratio chi-squared obtained are very small, so we unified the success rates of the four classes.

Table 7 shows the success rates of the different classes and where there were differences between the α and the likelihood.

Table 7. The different success rates of questions 3,5 and 6 in the different classes

	Class A	Class B	Class C	Class D
Question 3	**63**	**52**	**58**	22
Question 5	**71**	**93**	**83**	33
Question 6	42	**64**	**58**	26

The low success rates for class D can be explained by significant discipline problems that occurred.

3.6. Event handling with multiple actions

During lessons 7 through 9, the students were taught how to associate multiple actions with an event using two simple examples. In lesson 8 the students practiced this construct by implementing additional Braitenberg creatures. The students found the transition from one action to multiple actions difficult. Frequently, they duplicated the event and associated it with the second action, asking: "where

to put it?" or "what to do with…?". They tended to associate the position of a button with the direction of a movement, for example, they selected an event of touching the front button and then they were disappointed that the robot didn't move forward.

During the ninth lesson, the teacher solved questions together with the students. This facilitated bringing most of the students to a uniform level of understanding and, furthermore, helped clear up the difficulty of multiple actions associated with one event. They readily volunteered to come to the board to solve questions and were proud when they gave correct answers.

During the tenth lesson, the students learned about tap detection, clap detection and the music event blocks. They composed their own tunes and were enthusiastic about the feature of tapping and the clapping, which this facilitated practicing more advanced event handling.

3.7. The third questionnaire

The third questionnaire (administrated in lessons 14 and 15) involved questions regarding multiple actions per event. The success rates are shown in Table 8.

Table 8. Third questionnaire, % of students who gave correct answers for each question

Question no.	Success Rates
1	**75**
2	**61**
3	**68**
4	34
5	**61**
6	48

The success rates of questionnaire 3 were mixed: while the students were relatively successful on questions 1, 2, 3 and 5, they were less successful on questions 4 and 6, which required the students to complete a partial program and to create a program from scratch. Question 4 was hard since it required them to seek both an event and an action of a pair appropriate for the behavior that was given. The distractors made it difficult because they involved an incorrect ordering of an action before an event, incorrect direction of movement and irrelevant blocks. The fifth question investigated whether the students can perform near transfer of their existing knowledge.

The fifth question investigated whether the students can perform near transfer of their existing knowledge. The success rates in this question were not uniform for the four classes. The α obtained from the chi-squared test was 0.0467, which is greater than the likelihood ratio chi-Squared 0.0383; therefore, the null hypothesis could not be rejected.

The success rates of the four classes on question 5 of questionnaire 3 are presented in Table 9. The students achieved high success rates (>50%) in three of the four classes.

Table 9. The success rates of questionnaire number 3, question number 5 in the different classes

	Class A	Class B	Class C	Class D
Success Rates	**59**	**81**	**59**	44

The table shows that class B obtained very good success rates, while the success rate of classes A and D was much less though still greater than 50%. Class D did not achieve the desired success rate.

3.8. The bottom sensors and the line following algorithm

During lesson 11, the students explored the bottom sensors of the robot. The exploration activity included painting stripes of different colors and experimenting to see how the robot reacted to the different colors. The elementary physical principles of light sensitivity were explained. The exploration activity prompted some creativity on the part of the students, but it would have been preferable to supply the students with stripes in different colors, so that the students could readily identify the differences between the robot behavior to bright colors and dark colors. They kept asking: "Is it dark enough?" or "Is it bright enough?"

During lesson 12, the students received a detailed explanation of the bottom sensors and the differences between them and the front and back sensors. The students were a bit confused on this issue. The task required them to build a white track and a black track, and to execute different line-following algorithms. While the students enjoyed constructing lines using white and black tapes, they were confused by the execution of the algorithms. They kept asking questions regarding the bottom sensors block such as: "Should it be black?" or "What does the red frame mean?" The students invested much effort into building the tracks instead of attempting to understand the meaning of the bottom sensors event.

Lesson 13 started with a review on how the bottom sensors operate and how to use the bottom sensors event block in VPL. The students used the tracks they created during the previous lesson and once again implemented the line-following algorithms. Additionally, the students executed a program in which the robot had to stop at the edge of the table by sensing the edge with the bottom sensor. Again, the students had difficulties executing these algorithms and they needed more assistance about adjusting the bottom sensors.

During lesson 14 the students repeated the exercises they solved in the previous lessons; the repetition was helpful because it summarized the topic and tied up loose ends.

The learning difficulties might have been avoided if each event-action pair had been written and executed as a separate program before integrating them into a single concurrent algorithm.

3.9. The fourth questionnaire

The fourth questionnaire (administrated in lessons 17 and 18) contained questions on the line following algorithm and the bottom sensors. The success rates are shown in Table 10.

Table 10. Fourth questionnaire, % of students who gave correct answers for each question

Question no.	Success Rates
1	**51**
2	48
3	33
4	43
5	50
6	38

The success rates are significantly lower than the success rates of the the other questionnaires. The questions asked about advanced concepts: concurrent execution of event action pairs, implementing a relatively complex algorithm, and using the bottom sensors, which were confusing because of their similarities and differences with the front and back sensors.

In this questionnaire, there were no differences in the success rates on all question among the four classes. Therefore, the null hypothesis is rejected and we could accept the alternate hypothesis that the classes were similar.

3.10. The advanced mode lessons

The students were enthusiastic about learning and understanding the new blocks in the advanced mode and they were curious regarding the new possibilities that the blocks provided.

During lesson 16 the students studied the accelerometer events. In order to explain the functionality of the accelerometers, the teacher showed them a short video that illustrated how the robot was able to maintain its balance on a moving ball. The initial lesson on the accelerometers took place in their normal classroom, not in the computer lab. During the lesson they learned about left/right tilt and forward/backward tilt, and explored how to detection falling by using the accelerometers. Moreover, they explored how different angles can be identified by the accelerometers. The students loved to guess the number of the angles that the robot can detect. The robot was programmed to display a different color for color for each angle and the students grasped the idea rapidly.

In lesson 17 the students practiced exercises with the accelerometers. One problem we encountered was that in advanced mode the events are associated with states, a topic that was not taught, and this caused them to ask a lot of questions and to make mistakes. They solved the exercises quite easily and understood the functionality of the accelerometers.

During the lesson 18, the students learned about the timer event and the timer action. They found the concept exciting and learned how to use the timer to cause the robot to change its color and the direction of its motion after a period of time.

During the next two lessons, the students practiced with these blocks. They encountered some problems and were a little confused on how to write programs that used the timer event and action. It

was unnatural for them to put the blocks in different event-action pairs and they kept asking where to put the timer blocks.

During the last class they received an exercise that summarized the entire content of the syllabus. The exercise was not easy for the students and they didn't always volunteer to come to the board to help find a solution.

3.11. The teachers' focus group

The principal and the four teachers of the classes were very receptive to our suggested activities, because the school had been established only a few years previously and the staff was open to new initiatives. The teachers were cooperative and helped us understand the cognitive abilities and the affective aspects of the students. The focus group was held after the classes ended, so that the teachers would have a perspective on the entire course.

The first question was: What do they think about the content of the syllabus? The teachers agreed that the students understood the content of the lessons; they succeeded in their tasks and were able to answer the questionnaires. Moreover, they mentioned that the subject was "fun and cool for the students, and helped the students to bond and to perform the tasks." The teachers believed that the level of understanding students increased throughout the course and that they fully cooperated with us. As in every class, the students showed different levels of understanding, but they helped each other overcome the obstacles they encountered.

The second question was: How did the students profit from the course beyond learning the CS concepts? The teachers indicated that the students cooperated while working in groups. The students had to know how to share their work. The teachers indicated that while most of students were enthusiastic about the class, there were some who were not so excited. Unexpectedly, the teachers noted that most of the students who didn’t like the robotics class were boys, while the girls were more enthusiastic and curious.

The teachers added that the classes helped the students become more motivated about CS and robotics. The opportunity to work by trial and error helped them overcome difficulties and study the subject in a "fun way."

3.12. The project lesson

During the project lesson the students had to come up with their own ideas for programs they wanted to create. The first step was for them to think about the purpose of the program they wanted to create. The second step was to specify the program's steps, which required the students to analyze the purpose of the program and then to decompose it small steps of event-action pairs. We provided a worksheet with several ideas; however, they did not use these ideas and insisted on creating something of their own. The problem was that they wanted to create a program from scratch, but they didn’t look at it as an opportunity to make a program with a purpose; instead they looked at it as a just fun activity. The

result was that they created programs that showcased the VPL constructs, for example by using a large variety of blocks, but that had no purpose.

Most of the groups managed to create event-action pairs and to explain what the program performs when asked, but some used the advanced blocks incorrectly. For example, they used the states incorrectly, or used the timer event and action in the same pair, which is not meaningful. Most of the students included in their programs "cool" blocks (in their words), such as music, tapping and clapping. They tended to use buttons more than sensors, which are more fundamental in robotics. Most students managed to create a program that functioned correctly.

4 Discussion

The research goal was to characterize the learning outcomes of young second-grade students who took part in a CS through robotics course. We first discuss the findings from the observations and then discuss the achievement of the students in terms of the new taxonomy. Finally, we discuss the three conjectures and present the students' capabilities as measured by the questionnaires.

4.1. Observations

After the 21 lessons that included both frontal instruction and computer labs, the students appeared to like their experience with robotics. Most of the students seemed engaged throughout the lessons and were motivated to succeed in their assignments. They willingly participated in the class demonstrations and worked on the lab assignments, creating meaningful programs.

The robot was perceived by the students as an integral part of the learning, making their first experience with CS a positive one. The robot made CS more tangible, allowing the students to have hands-on interaction with the abstract concepts that they learned. The students tended to imagine that the robot had human qualities. They bonded with the robot, which helped them to overcome difficulties and to be even more engaged during the lessons. In particular, the Braitenberg creatures formed a bridge between the abstract and the tangible, allowing the students to implement complex event handling even with multiple actions.

4.2 Learning of CS concepts

The students had difficulties with the transitions between one event-action pair and several event-action pairs, and between one action per event and multiple actions per event. These difficulties were resolved up with more practice. As they learned more blocks, confusion arose as to which block should be used for which purpose. Furthermore, there was confusion between the bottom sensors and the horizontal sensors, which impaired the students' ability to understand the line following algorithm.

A partial explanation of these difficulties could be that they arose from the sharing of a robot by groups of three students, which resulted in friction within the groups. The students were not allowed to take the robots home, so they were not able to practice creating and executing programs on their own. The latest version of the VPL environment can be run in a software-only simulation, so that

programming can be practiced even in the absence of a physical robot.

When compared with previous work such as Bers et al. (2014), which only checked the success of the assignments that the students worked on in class, our work investigated their ability to go beyond what was presented within the framework of the class in order to demarcate what the students of this age can learn.

4.2. The taxonomy

Question 1 showed that the students are capable of correctly predicting the output of a given program. In all of the questionnaires most of the students answered this question correctly, in particular, the success rates were very high in the first and third questionnaires. This shows that even young students are able to analyze programs.

Question 2, which required the students to choose the program that gives rise to a given output, showed that the students are also capable of identifying a program that can give rise to a given output. Most of the students correctly answered this question in questionnaires 1 and 3, but had difficulties with questionnaire 2. This was probably due to premature testing of the students' capabilities, before they had enough experience with the blocks and with multiple event-action pairs. The low success rates for questionnaire 4 will be discussed in section 4.3.

The students encountered difficulties coping with questions of type 3, which required the students to watch two videos and to choose the statement that correctly described the difference between them. The low success rates were significant for questionnaires 1, 2 and 4. They found it difficult to identify behaviors shown in the videos and memorize them in order to answer the question. It is possible that if the two programs were given in addition to the two videos, the students would not have had to memorize the content of the videos and their success rates would have been higher.

The success rates of the students for questions of type 4 were high in both the first and second questionnaires, but were low in the third and the fourth questionnaires. This question required the students to complete a partial program in order to achieve a given output; this is the first type of question in the producing category. The results indicated that the students knew how to use the different blocks and they understood the functionality of the blocks, in addition to being able to analyze the given code, understand the functionality of the missing parts and how to use them to complete a program.

The success rates of the students were high for questions of type 5 except in the fourth questionnaire. This result is interesting because it means that the students were capable of using the knowledge they gained during the lessons in order to build new knowledge. This is consistent with near transfer of knowledge (Gick & Holyoak, 1980).

Question 6, which required the students to create a program from scratch when the goal of the program is given, showed low success rates in all of the questionnaires. While the students were able to write programs during class using trial and error, and with the help of the teaching staff, they found it difficult to write programs outside the context of the robot and the VPL environment. (Recall that the

questionnaires were administered offline.) Moreover, when asking the students to build a program of their own during the project lesson, the students were not able to give a purpose to the programs; instead, they just paired events and actions.

4.3. Interpreting the results in terms of the conjectures

The students' success in the class assignments where they wrote programs for the robot, together with the results of the first, second and third questionnaires, showed that the students are reaching relatively high levels of the taxonomy. This provides evidence that the students are capable of understanding and execute simple programs. In addition, the teaching staff observed the success of the students on the assignments. The students showed proficiency in using the relatively advanced construct of multiple actions per event. Given that event handling is considered to be a relatively advanced core concept of computer science, we conclude that the students were capable of understanding CS constructs.

On the other hand, when the students were asked about these concepts in the questionnaires or the worksheets (in particular in the fourth questionnaire), they encountered many difficulties and struggled to answer the questions and to describe the goal of a program. These difficulties also appeared during the project phase. The contrast between the performance in class and the difficulties with the questionnaires leads me to conclude that the students demonstrated the Jourdain effect: performance does not necessarily imply understanding.

The inability of the students to specify and implement their own projects showed that while the students were able to understand constructs, they were not capable of creating plans as defined by Soloway and Sphorer (1989).

5 Conclusions

Robotics activities enable even young students to learn basic CS concepts and they are capable of writing and running programs. Students performed well in answering questions on basic programming constructs and the VPL environment enabled the students to create programs graphically, thus overcoming the linguistic barriers to programming. Robotics activities can be successfully used with very young students to increase their interest and possibly motivation to become engaged with STEM in general and CS in particular.

However, our results showed that young students find it difficult to go from learning concepts and individual programming constructs to being able to create programs of more than a few lines. Another important conclusion is that students only functioned well when using the physical robot and with the help of the teaching staff.

Further research is needed in order to map CS concepts to the cognitive capabilities of students at various ages, in order to guide age-appropriate curriculum development for elementary schools.

Acknowledgements

We would like to thank Stella Khazina for assisting in the classroom throughout the entire course, and the teachers and principal of the school where the research was carried out.

References

Anderson, L. W., Krathwohl, D. R., Airasian, P., Cruikshank, K., Mayer, R., Pintrich, P., ... & Wittrock, M. (2001). *A taxonomy for learning, teaching and assessing: A revision of Bloom's taxonomy*. New York. Longman Publishing.

Artz, A. F., & Armour-Thomas, E.(1992). Development of a cognitive-metacognitive framework for protocol analysis of mathematical problem solving in small groups. *Cognition and Instruction*, 9(2), 137-175.

Anderson, M., McKenzie, A., Wellman, B., Brown, M., & Vrbsky, S. (2011). Affecting attitudes in first-year computer science using syntax-free robotics programming. *ACM Inroads*, 2(3), 51-57.

Armoni, M. (2012). Teaching CS in kindergarten: How early can the pipeline begin? *ACM Inroads*, 3(4), 18-19.

Barker, B. S., & Ansorge, J. (2007). Robotics as means to increase achievement scores in an informal learning environment. *Journal of Research on Technology in Education*, 39(3), 229-243.

Ben-Ari, M. (2013). *First Steps in Robotics with the Thymio Robot and the Aseba/VPL Environment*. https://aseba.wdfiles.com/local--files/en:visualprogramming/thymio-vpl-tutorial-en.pdf (last accessed 11 January 2018).

Ben-Bassat Levy, R., & Ben-Ari, M. (2015). Robotics Activities: Is the Investment Worthwhile? In *International Conference on Informatics in Schools: Situation, Evolution, and Perspectives*, Ljubljana, Slovenia. (pp. 22-31).

Bers, M. U., Flannery, L., Kazakoff, E. R., & Sullivan, A. (2014). Computational thinking and tinkering: Exploration of an early childhood robotics curriculum. *Computers & Education* 72, (pp. 145-157).

Biggs, J. B., & Collis, K. F. (2014). *Evaluating the quality of learning: The SOLO taxonomy (Structure of the Observed Learning Outcome)*. Academic Press.

Bloom, B. S., Engelhart, M. D., Furst, E. J., Hill, W. H., & Krathwohl, D. R. (1956). *Taxonomy of educational objectives, handbook I: The cognitive domain* (Vol. 19). New York: David McKay.

Braitenberg, V. (1984). *Vehicles: Experiments in synthetic psychology*. Cambridge, MA: MIT Press.

Bruce, K. B., Danyluk, A. P., & Murtagh, T. P. (2006). *Java: An Eventful Approach*. Pearson.

Brousseau, G. (2006). *Theory of didactical situations in mathematics: Didactique des mathématiques, 1970–1990 (Vol. 19)*. Springer.

Clements, D. 2002. Computers in early childhood mathematics. *Contemporary Issues in Early Childhood*, 3(2), 160-181. Available from:

http://www-tc.pbskids.org/read/brochure/powerpoint/Clements_Computers_Math.pdf. Accessed October 25, 2017.

Clements, D. H., & Sarama, J. (1997). Research on LOGO: A decade of progress. *Computers in the Schools*, 14(1-2), 9-46.

Clements, D., & Sarama, J. 2003. Strip mining for gold: Research and policy in educational technology—a response to "Fool's Gold". *Educational Technology Review*, 11(1), 7-69.

Druin, A., & Hendler, J. A. (Eds.). (2000). *Robots for kids: Exploring new technologies for learning*. Morgan Kaufmann.

Duncan, C., & Bell, T. (2015). A pilot computer science and programming course for primary school students. In *Proceedings of the 10th Workshop in Primary and Secondary Computing Education*, London, UK (39-48).

Duncan, C., Bell, T., & Tanimoto, S. (2014). Should your 8-year-old learn coding?. In *Proceedings of the 9th Workshop in Primary and Secondary Computing Education*, Berlin, Germany (60-69).

Fagin, B., & Merkle, L. (2003). Measuring the effectiveness of robots in teaching computer science. In *ACM SIGCSE Bulletin* 35(1), 307-311).

Fuller, U., Johnson, C. G., Ahoniemi, T., Cukierman, D., Hernán-Losada, I., Jackova, J., ... & Thompson, E. (2007). Developing a computer science-specific learning taxonomy. In *ACM SIGCSE Bulletin* 39(4), 152-170.

Gick, M. L., & Holyoak, K. J. (1980). Analogical problem solving. *Cognitive Psychology* 12(3), 306-355.

Glaser, B. (2017). *Discovery of grounded theory: Strategies for qualitative research*. Routledge.

Guzdial M., (2015) *Learner-Centered Design of Computing Education: Research on Computing for Everyone*, San Rafael, CA: Morgan & Claypool.

Hogg, D. W., Martin, F., & Resnick, M. (1991). *Braitenberg creatures. Cambridge: Epistemology and Learning Group*, MIT Media Laboratory.

Liao, Y. C., & Bright, G. W. 1991. Effects of computer programming on cognitive outcomes: a meta-analysis. *Journal of Educational Computing Research*, 7(3), 251-266.

Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., ... & Simon, B. (2004). A multi-national study of reading and tracing skills in novice programmers. In *ACM SIGCSE Bulletin* 36(4), 119-150.

Kaloti-Hallak, F., Armoni, M., & Ben-Ari, M. (2015). Students' attitudes and motivation during robotics activities. In *Proceedings of the 10th Workshop in Primary and Secondary Computing Education*, London, UK, 102-110.

Kay, J. S. (2011). Contextualized approaches to introductory computer science: the key to making computer science relevant or simply bait and switch? In *Proceedings of the 42nd ACM technical symposium on Computer science education*, Dallas, TX, 177-182.

Magnenat, S., Riedo, F., Bonani, M., & Mondada, F. (2012). A programming workshop using the

robot "Thymio II": The effect on the understanding by children. In *Advanced Robotics and its Social Impact*, Munich, Germany, 24-29.

Magnenat, S., Shin, J., Riedo, F., Siegwart, R., & Ben-Ari, M. (2014). Teaching a core CS concept through robotics. In *Proceedings of the 19th Conference on Innovation & Technology in Computer Science Education*, Uppsala, Sweden, 315-320.

Markham, S. A., & King, K. N. (2010). Using personal robots in CS1: experiences, outcomes, and attitudinal influences. In *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education*, Bilkent, Turkey, 204-208.

Martinez, C., Gomez, M. J., & Benotti, L. (2015). A comparison of preschool and elementary school children learning computer science concepts through a multilanguage robot programming platform. In *Proceedings of the 15th ACM Conference on Innovation and Technology in Computer Science Education*, Vilnius, Lithuania, (pp. 159-164).

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

Papert, S. 1980. Mindstorms: *Children, Computers, and Powerful Ideas, 2nd ed*. New York: Basic Books.

Pattis, R. E. (1981). *Karel the robot: A gentle introduction to the art of programming*. John Wiley & Sons.

Seiter, L., & Foreman, B. (2013). Modeling the learning progressions of computational thinking of primary grade students. In *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research*, 59-66.

J. Shin, R. Siegwart, and S. Magnenat. Visual Programming Language for Thymio II Robot. *Interaction Design and Children* (IDC), 2014.

Sierpinska, A. (2003). *Lectures on the Theory of Didactic Situations in Mathematics, Lecture 4*. http://annasierpinska.rowebca.org/pdf/TDSLecture%204.pdf (last accessed 11 January 2018).

Spohrer, J.C. & Soloway, E. (1986). Novice mistakes: Are the folk wisdoms correct?. *Communincations ACM* 29(7), 624-632.

Sullivan, A., & Bers, M. U. (2016). Robotics in the early childhood classroom: learning outcomes from an 8-week robotics curriculum in pre-kindergarten through second grade. *International Journal of Technology and Design Education*, 26(1), 3-20.

Tedre, M. & Denning, P.J. (2016). The long quest for computational thinking. In *Proceedings of the 16th Koli Calling International Conference on Computing Education Research*, 120-129.

Wyeth, P. (2008). How young children learn to program with sensor, action, and logic blocks. *The Journal of the Learning Sciences*, 17(4), 517-550.

Pre-Service Information Technologies Teachers' Views on Computer Programming Tools for K-12 Level

Serhat Altıok[1]

Erman Yükseltürk[1]

[1]Kırıkkale University

DOI: 10.21585/ijcses.v2i3.28

Abstract

The purpose of the study is to analyze pre-service IT teachers' views on a five day seminar which is related to current methodologies and tools in K-12 computer programming education. The study sample consisted of 44 pre-service IT teachers who study as 3rd or 4th undergraduate program at Department of Computer Education and Instructional Technology in 21 different universities. The data is collected through a Students' Perceptions about Kid's Programming Language Questionnaire consisting of 27 five-point Likert-type items, grouped under three factors. The collected quantitative data were analyzed using descriptive statistics such as means, standard deviations. The results of the study indicated that almost all visual programming tools have positive effects on students' views, Small Basic is not as effective as other tools. It could be concluded that Small Basic tool is text-based in contrast to the other block-based features.

Keywords: Programming Education, Visual Programming Tools, Pre-Service IT Teachers, Scratch, Small Basic, Alice, App Inventor

1. Introduction

In recent years, best practices of technologies have found new audiences with increasingly children from smartphones and tablet computers to electronic learning toys (Bers, Flannery, Kazakoff, & Sullivan, 2014). Therefore, the children in the 21st century children need to be versatile and adaptable not only modern and future technologies but also need to improve the ability of understand and work with these technologies (Saeli, Perrenet, Jochems, & Zwaneveld, 2011). In other words, it is expected that today's children have the knowledge and skills about these technologies and also use them

effectively in their life (Günüç, Odabaşı & Kuzu 2013). In addition to these knowledge and abilities, today's children are expected to have basic skills such as critical thinking, analyzing and synthesizing ability, some requirements related to the rapid development of technology such as information literacy, media literacy, technology literacy and code literacy etc. and personal qualities work collaboratively, being innovative and being productive (Kay & Greenhill, 2011). Programming is one of the common techniques that provides students to developing these knowledge, skills, requirements and competencies for solving real-world problems of the 21st century (Grover & Pea, 2013). There are several research studies in the literature which demonstrate the importance of programming since it enables children to become active producers of interactive digital environments and its positive effects (Fesakis & Serafeim, 2009; Kalelioğlu & Gülbahar, 2014) on academic success (Fessakis, Gouli, & Mavroudi, 2013), problem solving performance (Fesakis & Serafeim, 2009) and building children's computational thinking skills etc. (Bers et al., 2014; Brennan & Resnick, 2012).

Despite the benefits and importance of computer programming, it is considered to be difficult to master and understand the core concepts. The reason of this consideration is programming performed in several steps such as generate a solution to a problem, reflect on how to communicate the solution to the machine and using syntax and grammar through an exact way of thinking (Szlávi & Zsakó, 2006). In other words, programming performed in three steps (Pears et al., 2007): problem solving, learning a particular programming language and code/system production. Most students have difficulties in these steps of programming (Lister et al., 2004; Robins, Rountree, & Rountree, 2003). There are also numerous difficulties for students in programming learning and teaching in the literature (Du Boulay, 1986):

- Orientation (What programming is useful for and what the benefits to learn to program are)
- The notional machine (Understanding the general properties of the machine and how the behavior of the physical machine relates to the notional machine)
- Notation (Includes the problems of aspects of the various formal languages such as syntax and semantics)
- Structures (The schemas or plans that can be used to reach small-scale goals (e.g., using a loop))

Moreover, studies found in literature show that many problems in learning programming originate from abstract and complexity of the concepts such as variables, loops, arrays, functions, and syntax of programming languages. These difficulties may become barriers for learning programming skills (Ozoran, Cagiltay, & Topalli, 2012), especially for novices.

Actually, all programmers are novice at the beginning, and it has to know that learning to programming is hard and has to overcome from a wide range of difficulties and deficits (Winslow, 1996). However, turn a novice into an expert programmer takes roughly 10 years (Winslow, 1996) and this continuum has five breakdowns into stages (Dreyfus, 1986): novice, advanced beginner,

competence, proficiency, and expert. Just because an expert must be mastered on five specialties (Du Boulay, 1986): general program knowledge, hardware components and their relationship with programs, syntax and semantics of a particular programming language, structures (e.g., schemas, plans), and approaches or theories (that includes continuums such as planning, developing, testing and debugging etc.)

Unlike the experts, novices limited superficially knowledge, lack detailed mental models, fail to apply relevant knowledge, and approach programming ‘‘line by line’’ rather than using meaningful program ‘‘chunks’’ or structures (Winslow, 1996). Novice programmers hold misunderstanding and misconceptions (Saeli et al., 2011). Thus, novices’ incomplete prior knowledge such as misconceptions, deficits in planning, developing and testing of code can be a source of errors, and more (Spohrer & Soloway, 1989). This information shows that students should be coached in the process of programming and teaching programming in early steps in personal training should be provided with facilitating methods and tools.

1.1. Programming Education

A simple definition of programming is the process of writing, testing, debugging/troubleshooting, and maintaining the source of code of computer programs (Wikipedia, 2017). Another definition of programming is the process of developing and implementing various sets of instructions to enable a computer to perform a certain task, solve problems, and provide human interactivity (ECDL Foundation, 2015). Similarly, Moström (2011, p9) defined that “programming is the act of understanding a problem, formulating a solution, and writing down the solution in such a way that a computer can use the solution to solve the problem”. Saeli et al. (2011) analyzed what the reasons to teach programming and their results are enhancing students’ problem solving skills and offering the students a subject, which includes aspects of different disciplines; use of modularity and transferability of the knowledge/skills; and the opportunity to work with a multi-disciplinary subject.

Programming education that has been realized and researched extensively through different methods, languages, tools and methodologies at different levels from primary education to university level have been increased its importance more and more every day. In recent years, there have been a growing number of countries which are focusing their Information and Communications Technology (ICT) curricula on developing students’ computer programming and coding skills that facilitates building higher-order thinking skills. For example, computer programming and coding became an important part of the curriculum in 12 Europe countries: Bulgaria, Cyprus, Czech Republic, Denmark, Estonia, Greece, Ireland, Italy, Lithuania, Poland, Portugal and the UK (Johnson et al., 2014). Similarly, eight countries integrate coding in the ICT curriculum, some countries such as United Kingdom, Estonia, Greece, and Lithuania integrate programming not only in the general ICT course, but also as a specific

standalone course in primary curricula (Johnson et al., 2014). Despite most of the countries introducing computing in K-12 curriculum as a whole, some countries in Europe have been applied introducing computing in either K–9 or grades 10–12 (Heintz, Mannila, & Färnqvist, 2016).

Moreover, well-known efforts in the US are the "Computer Science Principles" that aims to develop effective high school computing curricula enacted in 10,000 high schools taught by 10,000 well-prepared teachers by 2016, "The Beauty and Joy of Computing" that exposes students to the beauty and joy of programming by engaging them in meaningful projects using the Snap! Programming language, and "Code.org" that is a high school course with lessons and programming projects (Angeli et al., 2016; Astrachan, Briggs, Diaz, & Osborne, 2013; desJardins, 2015). The main purpose of introducing computing in primary education is to produce thinkers, as opposed to coders (Repenning, Basawapatna, & Escherle, 2016). In spite of the fact that the coders write codes in any programming languages for solving any problems through produce software that can be divide into application and system, thinkers develop patterns to solve all problems of that type instead of solving any problem by produce generalizable solutions (Selby & Woollard, 2013).

In Turkey, primary education involves core and track subjects at first and second level in primary education such as mathematics, science, social science, language and communication (Ministry of Education, 2012; Sağlam, 2014). Although these lecture-based subjects provide to students knowledge and skills (e.g. thinking, understanding and reasoning), they need to developed higher-order thinking skills such as critical, logical, reflective, metacognitive, and creative thinking (King, Goodson, & Rohani, 2010). For improve these skills, not only should core subjects should be integrated with each other but also Information Technologies (IT) lesson with properties such as student-centered, problem-based, or project-based must be utilized more effectively. Because of this requirement, Ministry of National Education of Turkey initially updated to Information Technologies course as Information Technologies and Software course in 2012, after defined IT lesson that was an elective subject as a compulsory subject at 5th and 6th grade levels of primary education in 2013. Moreover, a regulatory commission was formed by the ministry for include more coding training into the IT lesson curriculum that is compulsory at 5th and 6th grades and elective at 7th and 8th grade for developing algorithmic and computational thinking skills in students. In addition to public institutions, non-governmental organizations have also implemented various projects and activities in order to provide programming and software skills to the students (e.g. Informatics Association of Turkey (IAT) organized an event entitled "Computer Programming is as Easy as Pie " in May, 2014). Association of Information Technology Educators have been realized numerous organizations that including coding, robotics and physical programming trainings to teachers of Information Technologies for bringing to students more programming skills (e.g. Manisa City is Coding, Antalya City is Coding etc.) in recent years. These organizations have been carried through the contributions of the academicians in the departments of Computer Education and Instructional Technology of the universities in these cities. Additionally, academicians in this department realized numerous seminars and workshops through the

agency of support from public institutions to introduce alternative pedagogic approaches and visual programming tools that are used for teaching programming to students by teachers or pre-service teachers in recent years (e.g. "How to Teach Programming to Children" seminar and “Programming my Own Game” seminar, CEIT, Kırıkkale University). Besides all these institutional efforts, thousands of students from all grades of education from primary school to university participate individually web-based organizations such as “All Can Code”, “Hour of Code”, “Code Monkey” etc.

1.2. New Applications for Programming Education

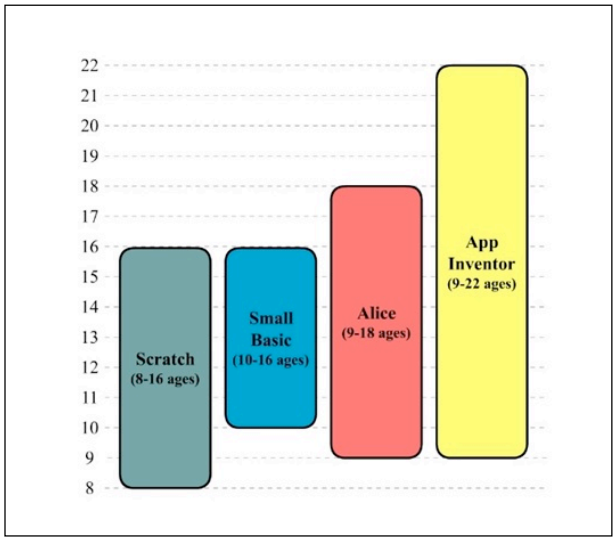
Visual programming affect novice programmers' performance and require them to manipulate visual elements to formulate and test of problem (Gouws, Bradshaw, & Wentworth, 2013; Maloney, Resnick, Rusk, Silverman, & Eastmond, 2010). In recent years, there are several visual programming tools allows users who has limited or no programming background to create interactive media-rich projects such as games, simulations, and animations. These tools have demonstrated their particular benefits to assist learning programming and problem solving (Lye & Koh, 2014), and help novice programmers to construct their programs and understand the process of program execution (Kelleher & Pausch, 2005). In brief, visual programming tools help novice programmers to improve programming skills and allows creating and demonstrating digital artifacts through problem solving strategies (Lye & Koh, 2014). Therefore, many visual programming tools available provide children to develop different types programs such as games, animations, interactive stories, mobile applications, or robotic applications. Researchers identified 113 different visual programming tools in many different types in the literature such as block-based, text-based or tile-based.

- Block-based visual programming tools (e.g., Scratch) allow users to construct scripts by dragging-and-dropping code blocks and provide visual feedback to comprehend how code blocks work (Maloney et al., 2010).
- Text-based visual programming tools (e.g., Small Basic) provide a simplified programming environment with syntax highlighting and code completion facility (Microsoft, 2017).
- Tile-based visual programming tools (e.g., Kodu Game Lab) enable users to create and play video games and animated stories through placement of tiles in a meaningful sequence (Fowler, Fristoe, & MacLaurin, 2012).

Furthermore, these tools are very different from the various features (e.g. purpose of use, age level, level of difficulty, whether or not paid, pedagogical effectiveness and platform type). Therefore, choosing the most effective and purposeful appropriate visual programming tool is directly related to have deep knowledge about features of these tools. In this study, researchers selected Scratch, Small Basic, Alice, and App Inventor because of their different properties such as types of product platform, coding styles, and 2D/3D structures. Another reason for researchers to choose these visual programming tools is age ranges proposed by the person (s), institution (s) or organization (s) that

developed these tools are appropriate. Appropriate age ranges of visual programming tools that recommended by the official producer are shown in Figure 1 below (Microsoft, 2017; MIT, 2017b).

Figure 1. The Appropriate Age Ranges Recommended by the Official Producer of Visual Programming Tools



Although research groups that produce visual programming tools have made suggestions that they are more appropriate at specified age ranges, this age range can be expanded in the direction of need or level of knowledge and use as bidirectional (reducing lower age limit or increasing upper age limit).

1.2.1. Scratch

Scratch (https://scratch.mit.edu/) is a free educational programming tool that was developed by the Lifelong Kindergarten group at the Massachusetts Institute of Technology (MIT) Media Laboratory. The Scratch project began in 2003, and its software and website were publicly launched in 2007. Nowadays, it hosts over 15 million shared projects and almost 80 million comments have been posted

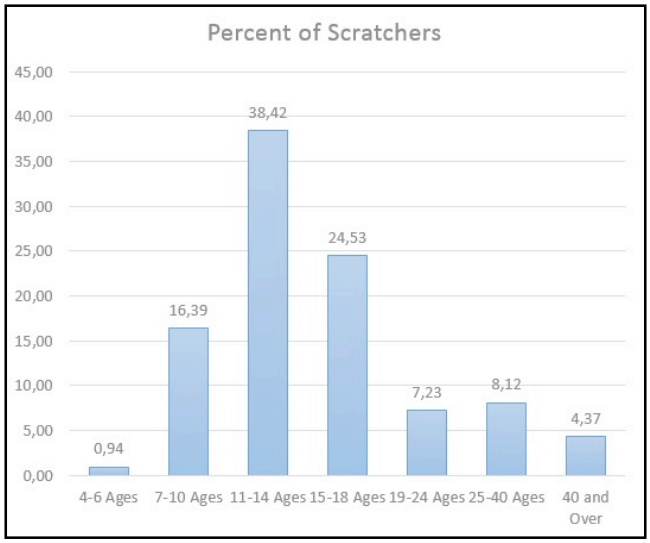


Figure 2. The Distribution of Scratchers' Ages

about projects by 12 million registered users (MIT, 2017b). The distribution of users' registration age is shown in Figure 2 below.

Scratch provides users create games, animations and simulations by dragging and dropping instead of coding that requires understanding or knowing some concepts such as variables, loops, arrays or functions. Because visual and enjoyable, scratch is preferred by especially young people (MIT, 2017b).

1.2.2. Small Basic

Small Basic (http://smallbasic.com/) is a free educational programming tool that was announced in October 2008, and the first stable version was released in July 2011 by Microsoft. This tool makes programming easy, approachable and funny because it supports conditional branching, loop structures, variables which are weakly typed and dynamic, and subroutines for event handling. Small Basic has two important components: coding screen that allows using codes very simply and a library that provides rich and engaging set of components. This tool also associated with Integrated Development Environment (IDE) which consists of code editor, automation tools and debugger that provides facilitating to software development (Microsoft, 2017).

1.2.3. Alice

Alice (http://www.alice.org) is a free and object-based educational programming tool that was developed on the programming language Python (http://www.python.org) by the Stage 3 Research Group at Carnegie Mellon University led by Randy Pausch in 1997. This tool makes programming easy to learn and create an animation, an interactive game, or a video by drag and drop graphic tiles. By manipulating the objects, Alice also allows to see running of animation, game or a video and facilitates to understand the relationship between programming and objects (Carnegie Mellon University, 1997).

1.2.4. App Inventor

MIT App Inventor (http://appinventor.mit.edu/) is a free and open-source educational programming tool that for designing and building mobile applications that can run on Android devices. App Inventor uses simple graphical interface that very similar to Scratch, which allows drag and drop building to create a basic, fully functional application within an hour or less. This tool was developed by the team was led by Hal Abelson, Mark Friedman, Eric Klopfer and Mitchel Resnick in March 2012. Nowadays, it hosts over 12 million applications have been built by 4 million registered users in 195 countries (MIT, 2017a).

2. Method

This is a descriptive study conducted with a quantitative basis. Descriptive studies, as the name implies, are carried out to describe the characteristics and views of the studied subject (Fraenkel & Wallen, 2009). In this study, the pre-service IT teachers' views were analyzed about computer programming teaching tools especially for K-12 level.

2.1. Participants

The sample of this study was selected from the participants who attended a five day seminar program at a university in Turkey. They were 44 pre-service IT teachers who study as 3rd or 4th undergraduate student at Department of Computer Education and Instructional Technology in 21 different universities.

Table 1. The characteristics of the participants

	N	%
Gender		
Male	19	43.18
Female	25	56.82
Weekly Hours of Computer Use		
0-2 hours	9	20.46
3-5 hours	12	27.27
6-8 hours	23	52.27
Computer Programming Knowledge		
Low	21	47.73
Intermediate	17	38.63
High	6	13.64

As it is showed in Table 1, nineteen (43.18%) of them were male and twenty five (56.82%) of them were female. 79.54% of the pre-service IT teachers use computer more than 3 hours in one day. In regard to computer programming knowledge, almost half of the participants (48%) rated their programming knowledge level as low.

2.2. Settings

This is a seminar program for pre-service IT teachers about alternative methods and tools in computer programming for K-12. It was supported by Scientific Meetings Grant Programs. Throughout the program, several seminars were organized to present pre-service IT teachers with pedagogical

information specifically for programming instruction for elementary school students and to demonstrate the practical use of current tools used to teach programming to students. The program was started with the introduction of basic pedagogical concepts that have an important place in the education of children and continued with pedagogical examination of how to teach children programming. Later, the tools used for programming teaching were described and the four most common tools used in teaching programming to children in recent years were mentioned. Basic programming principles with Scratch and Small Basic, 3D graphics programming with Alice, Android based mobile programming with App Inventor were discussed with practical examples at computer laboratories. Eight academicians from different universities participated as educators to give seminars for the program. Participants throughout the program were hosted by the University and all costs of the participants (e.g. road, accommodation and meals) were covered.

2.3. Instrumentation

To collect relevant data in this study, researchers used quantitative method. The following instrument helped us to collect quantitative data: Students' Perceptions about Kid's Programming Language Questionnaire (SPKPL-Q). It was developed by Akcay (2009) to obtain the students' perceptions about Small Basic. In this study, it was adapted for Scratch, Alice and App Inventor in addition to Small Basic. It is a 5-point Likert-type scale, consisting of 27 items, grouped under three factors (Perceived Motivation, Perceived Usefulness and Perceived Ease of Use). The Cronbach-Alpha reliability coefficient of the scale was found to be between 0.806 and 0.865.

2.4. Data Collection and Analysis

In the first day of the seminar program, the participants were mentioned about the pedagogy of programming education, programming tools and alternative methodologies. Later, the participants were taught about Scratch, Small Basic, Alice and App Inventor interface, usage and developed applications. At the end of seminars, researchers collected the quantitative data through the questionnaire which included four programming tools (Scratch, Small Basic, Alice and App Inventor). During analyzing of the collected data, the descriptive statistics such as mean and standard deviations of pre-service IT teachers' views about four programming tools were calculated based on the SPKPL-Q scale scores. Also, one way analysis of variance test (ANOVA) was conducted to test the mean differences of pre-service IT teachers' views about four programming tools. ANOVA was considered to be appropriate for the analysis of the data in the study because there is an analysis method which is used to test whether the difference between the averages of two or more unrelated samples is significantly different from zero (Büyüköztürk, 2004). Before the analysis of results, the assumptions of ANOVA have been tested. Each group has a normal distribution and the variances of the groups are homogenized ($p>0.05$).

3. Results

The findings of this study related with pre-service IT teachers' views about four programming tools are given in the Table 2 and Table 3. According to the Table 2, the percentage strongly agree or agree of pre-service IT teachers' views about Scratch programming in regard to perceived motivation, perceived usefulness and perceived ease of use were 92.2, 94.7 and 92.6 respectively. The percentage strongly agree or agree of pre-service IT teachers' views about Small Basic programming in regard to perceived motivation, perceived usefulness and perceived ease of use were 77.4, 77.6 and 55 respectively. The percentage strongly agree or agree of pre-service IT teachers' views about Alice programming in regard to perceived motivation, perceived usefulness and perceived ease of use were 88.1, 90.5 and 87.1 respectively. The percentage strongly agree or agree of pre-service IT teachers' views about App Inventor programming in regard to perceived motivation, perceived usefulness and perceived ease of use were 92.8, 96.2 and 88.2 respectively. According to the results, the pre-service IT teachers' views about programming tools were generally positive. The lowest ratio related to percentage of pre-service IT teachers' views about programming tools were Small Basic programming.

Table 2. The percentages of pre-service IT teachers' views on programming tools

	Scratch					Small Basic					Alice					App Inventor				
	SD	D	N	**A**	**SA**	SD	D	N	**A**	**SA**	SD	D	N	**A**	**SA**	SD	D	N	**A**	**SA**
	%	%	%	**%**	**%**	%	%	%	**%**	**%**	%	%	%	**%**	**%**	%	%	%	**%**	**%**
Perceived Motivation																				
Interest / Enjoyment	0.76	1.52	3.79	**21.97**	**71.97**	3.79	6.06	21.21	**34.85**	**34.09**	2.27	2.27	2.27	**32.58**	**60.61**	1.52	1.52	5.30	**21.97**	**69.70**
Perceived Competence	2.27	0.00	4.55	**31.82**	**61.36**	0.00	6.82	6.82	**54.54**	**31.82**	4.55	0.00	6.82	**36.36**	**52.27**	0.00	0.00	0.00	**36.36**	**63.64**
Willingness	4.54	3.03	1.51	**29.54**	**61.36**	3.79	10.61	14.39	**33.33**	**37.88**	5.30	3.79	6.82	**31.82**	**52.27**	5.30	3.03	0.76	**28.79**	**62.12**
Participation	0.00	2.27	6.82	**37.50**	**53.41**	3.41	3.41	18.18	**39.77**	**35.23**	1.14	1.14	11.36	**38.64**	**47.73**	1.14	2.27	7.95	**30.68**	**57.95**
Average				**30.21**	**62.03**				**40,62**	**34,76**				**34.85**	**53.22**				**29.45**	**63.35**
Perceived Usefulness																				
Work More Quickly	2.27	2.27	1.14	**26.14**	**68.18**	2.27	3.41	15.91	**32.95**	**45.45**	3.41	1.14	2.27	**29.54**	**63.64**	3.41	0.00	1.14	**30.68**	**64.77**
Job Performance	1.14	0.00	3.41	**31.82**	**63.64**	5.68	4.55	7.95	**39.77**	**42.04**	2.27	2.27	3.41	**35.23**	**56.82**	1.14	1.14	1.14	**32.95**	**63.64**
Increase	0.0	0.0	2.2	**36.3**	**61.3**			13.6	**45.4**	**34.0**	2.2	0.0		**34.0**	**56.8**	0.0	0.0	2.2	**38.6**	**59.0**

Makes Job Easier	0.00	2.27	2.27	**38.64**	**56.82**	0.00	11.36	15.91	**47.73**	**25.00**	0.00	2.27	13.64	**34.09**	**50.00**	0.00	2.27	0.00	**40.91**	**56.82**
Useful	0.00	1.14	6.82	**34.09**	**57.95**	1.14	5.68	12.50	**42.04**	**38.64**	0.00	0.00	5.68	**43.18**	**51.14**	0.00	0.00	3.41	**38.64**	**57.95**
Average				**32.77**	**61.93**				**40.71**	**36.93**				**34.66**	**55.87**				**35.61**	**60.61**
Perceived Ease of Use																				
Easy to Learn	1.14	1.14	1.14	**39.77**	**56.82**	11.36	17.04	26.14	**23.86**	**21.59**	1.14	4.55	10.23	**46.59**	**37.50**	0.00	3.41	4.55	**45.45**	**46.59**
Easy to Use	2.27	2.27	4.55	**25.00**	**65.91**	4.55	6.82	27.27	**43.18**	**18.18**	0.00	2.27	0.00	**50.00**	**47.73**	0.00	0.00	9.09	**31.82**	**59.09**
Easy to Become Skillful	0.00	0.00	6.82	**40.91**	**52.27**	6.82	9.09	36.36	**25.00**	**22.73**	0.00	4.55	11.36	**43.18**	**40.91**	0.00	2.27	9.09	**38.64**	**50.00**
Clear and Understandable	4.09	5.45	0.91	**29.54**	**60.00**	6.36	10.45	17.78	**30.91**	**34.54**	1.82	8.64	7.27	**35.45**	**46.82**	3.18	5.91	9.55	**28.18**	**53.18**
Average				**33.81**	**58.75**				**30.74**	**24.26**				**43.81**	**43.24**				**36.02**	**52.22**

Note. SD: Strongly Disagree, D: Disagree, N: Neutral, A: Agree, SA: Strongly Agree.

According to the Table 3, the overall means of pre-service IT teachers' views about Scratch programming in regard to perceived motivation, perceived usefulness and perceived ease of use were 4.49, 4.55 and 4.45 respectively. The overall means of pre-service IT teachers' views about Small Basic programming in regard to perceived motivation, perceived usefulness and perceived ease of use were 3.98, 4.04 and 3.54 respectively. The overall means of pre-service IT teachers' views about Alice programming in regard to perceived motivation, perceived usefulness and perceived ease of use were 4.33, 4.42 and 4.24 respectively. The overall means of pre-service IT teachers' views about App Inventor programming in regard to perceived motivation, perceived usefulness and perceived ease of use were 4.51, 4.55 and 4.36 respectively. The post-hoc test results indicated that there is a significant differences about pre-service IT teachers' views about four programming tools ($p<0.05$). In other words, pre-service IT teachers' views were generally positive, but, the means of their views related to Small Basic programming were lower when comparing with other programming tools.

Table 3. The mean differences of pre-service it teachers' views on programming tools

	Scratch		Small Basic		Alice		App Inventor			
	M	**SD**	**M**	**SD**	**M**	**SD**	**M**	**SD**	***F***	***p***
Perceived Motivation										
Interest / Enjoyment	4.63	0.54	3.89	0.83	4.47	0.60	4.57	0.58	11.99	0.00*
Perceived Competence	4.50	0.79	4.11	0.81	4.32	0.96	4.64	0.49	3.70	0.01*
Willingness	4.40	0.62	3.91	0.91	4.22	0.70	4.39	0.63	4.46	0.00*
Participation	4.42	0.59	4.00	0.85	4.31	0.69	4.42	0.66	3.51	0.02*
Overall Mean	**4.49**	**0.64**	**3.98**	**0.85**	**4.33**	**0.74**	**4.51**	**0.59**		
Perceived Usefulness										
Work More Quickly	4.56	0.69	4.16	0.83	4.49	0.66	4.53	0.67	2.96	0.03*
Job Performance	4.57	0.56	4.08	0.89	4.42	0.69	4.57	0.56	4.96	0.00*
Increase Productivity	4.59	0.54	4.05	0.94	4.43	0.82	4.57	0.55	5.23	0.00*
Effectiveness	4.57	0.62	4.00	0.96	4.41	0.84	4.55	0.63	5.03	0.00*
Makes Job Easier	4.50	0.66	3.86	0.93	4.32	0.80	4.52	0.63	7.02	0.00*
Useful	4.49	0.54	4.11	0.85	4.45	0.55	4.55	0.52	4.23	0.01*
Overall Mean	**4.55**	**0.60**	**4.04**	**0.90**	**4.42**	**0.73**	**4.55**	**0.59**		
Perceived Ease of Use										
Easy to Learn	4.50	0.60	3.27	1.16	4.15	0.72	4.35	0.61	20.54	0.00*

Easy to Use	4.50	0.88	3.64	1.01	4.43	0.62	4.50	0.66	11.91	0.00*
Easy to Become Skillful	4.45	0.63	3.48	1.15	4.20	0.82	4.36	0.75	11.73	0.00*
Clear and Understandable	4.36	0.54	3.77	0.86	4.17	0.59	4.22	0.58	6.57	0.00*
Overall Mean	**4.45**	**0.66**	**3.54**	**1.05**	**4.24**	**0.69**	**4.36**	**0.65**		

Note. M: Mean, SD: Standard Deviation, *: $p<0.05$

4. Discussion

Programming is the most functional way for supporting to developing higher-order thinking skills and algorithmic problem-solving skills (Grover & Pea, 2013; Weintrop et al., 2016). There are numerous programming languages in the literature such as Python, C, C++, C#, Java, JavaScript, PHP, Assembly language, Visual Basic .NET, Perl, Delphi, Ruby, Scala, Haskell, Swift etc. (Pierce, 2002). These structured (conventional) programming languages are widely adopted by educators to teach general purposes of any programming (Pears et al., 2007; Robins et al., 2003; Xinogalos, 2012). The first step in all programming languages is to learn or teach what the core elements such as condition, array, loop, variable, constant, and functions are, what they do, and how they are used. Studies found in literature showed that many problems in learning programming originate from using of these core elements (Choi, 2012). Since learning these core elements is difficult and tedious process, various projects and activities are organized with the support of non-nonprofit companies, non-governmental organizations and governments in order to make students love programming by making coding easy and fun. One of these widespread efforts is the use of visual programming tools (e.g. Scratch, Alice) in programming education. In this study, a seminar program was organized about teaching alternative methods and tools in computer programming for K-12 level in order to reach similar aims. The pre-service IT teachers' views who attended this seminar program were analyzed based on four visual programming tools and the results showed that they have positive views on the use of these programming tools in programming education in terms of motivation, usefulness and ease of use.

In the last decade, there have been developed several visual programming tools that make students more effective producer through some features such as simplified syntax, drag and drop ability to compose programs, immediate execution of commands. Also, these visual programming tools help novice programmers to improve programming skills and allow creating and demonstrating digital artifacts through problem solving strategies (Lye & Koh, 2014). In other words, visual programming environments have been developed like Scratch, Small Basic, Alice, Lego Mindstorm, in order to make them more compatible to information technology beginners to minimize the learning disabilities and difficulties of the programming. Similarly, almost all visual programming tools have positive effects on pre-service IT teachers' views in this study.

Computer programming skills and learning these skills become more important in the 21st century. However, programming lessons which are given by traditional methods do not attract to students

attention and various problems occur. These problems cause to check the teaching method used in the programming education. Alternative visual programming tools have been developed to minimize the problems in programming education. Even though the text-based programming is still common method for teaching programming skill, it has several drawbacks. Similarly, Small basic is more unsuccessful than other visual programming tools such as Scratch, Alice, and App Inventor since it contains tasks likewise in the conventional programming in this study. The results showed that scripting has a negative effect on the pre-service teacher's attitudes towards programming when comparing visual programming tools.

5. Conclusion

Programming is becoming increasingly a key competence which will have to be supported generating computer-based solutions for problems such as program, application, animation, simulation, or game by all students since it improves computational thinking skills as well as high-level thinking skills such as developing creative, critical, strategic, analytical, multidimensional, solution-focused. Developing individuals who have these high-level thinking skills that cannot be developed in a short time and develop as experience grows depends on the provision of programming education as long-term and product-focused to individuals from a young age. However, different requirements of programming languages such as syntax, constructs etc. cause problems in developing individual products. Because of these problems, individuals drop out from their programming education. Instructors prefers visual programming tools in programming education to overcome these problems. Especially at primary education level, it is even more difficult to learn the text codes and use the syntax properties of the programming language without any problems that should be written in languages other than native languages because of the low level of foreign language skills at small age levels. In addition to developing the solutions to problems easier with visual programming, independently testable of code blocks for sub-problems that provided with tools like Scratch provides more effective process on develop algorithms for solving sub-problems for primary students.

In recent years, various summer camps, seminars, educational and social projects have been challenged in order to bring several skills to the students at an early age. One of these has been studied in this study. The aim of this seminar program is to develop the knowledge and skills of IT teacher candidates to update the information about programming education for elementary school students in particular, to demonstrate the practical use of current tools used to teach programming to students, to prepare classroom teaching activities using these applications and to adapt them to laboratory activities. This study analyzed pre-service IT teachers' views on this five day seminar which is related to current methodologies and tools in K-12 computer programming education. The research results showed that pre-service IT teachers have positive views on the use of visual programming tools in programming education. However, it is noteworthy that the views on Small Basic are less positive than the other three tools. According to this information, when pre-service IT teachers begin their

professional career, it may be less likely to prefer Small Basic because of its text-based programming requirements. Nevertheless, it is important that pre-service IT teachers will use even three of the four tools which are trained to use as a trainer. If pre-service IT Teachers are taught a greater variety of visual programming tools then they can specialize in the use of one specific tool that can be more appropriate and effective for them. For this reason, pre-service IT teachers should be provided numerous trainings that includes alternative tools and methodologies.

All of these conditions also drives the development of several educational programming tools especially for novices and young students. On the contrary, we have to think that the availability of software programming environments is not enough for the utilization of the learning potential of programming. Experimentally validated teaching/learning approaches, documented best practices, learning resources, curriculum standards, professional development and support for teachers are also needed (Fessakis et al., 2013). In addition, pre-service teachers want to be aware of advanced technology and current pedagogical information and they need training to improve themselves. Increasing the number of such events will be important in terms of updating teacher candidates' information and informing them of new developments. Some potential limitations of this study also should be taken into consideration while discussing the results since only four visual programming tools were analyzed in a seminar program. The study population consisted of only 44 pre-service IT teachers attending in this seminar, which limits the generalizability of the results. Extending the population to various activities, programs and universities could produce different results.

Acknowledgments

This study was conducted with financial support of The Scientific and Technological Research Council of Turkey (Project No: Scientific Meetings Grant Programmes-2229-2016/1).

References

Akçay, T. (2009). Perceptions of students and teachers about the use of a kid's programming language in computer courses. Unpublished MS Thesis. Middle East Technical University, Ankara, Turkey.

Angeli, C., Voogt, J., Fluck, A., Webb, M., Cox, M., Malyn-Smith, J., & Zagami, J. (2016). A K-6 computational thinking curriculum framework: implications for teacher knowledge. *Educational Technology & Society,* 19(3), 47-58.

Astrachan, O., Briggs, A., Diaz, L., & Osborne, R. B. (2013). CS principles: development and evolution of a course and a community. *Paper presented at the Proceeding of the 44th ACM technical symposium on Computer science education.* Retrieved June 15, 2018, from https://doi.org/10.1145/2445196.2445382

Bers, M. U., Flannery, L., Kazakoff, E. R., & Sullivan, A. (2014). Computational thinking and tinkering: Exploration of an early childhood robotics curriculum. *Computers & Education, 72*, 145-157. Retrieved June 15, 2018, from http://doi.org/110.1016/j.compedu.2013.1010.1020

Brennan, K., & Resnick, M. (2012). New frameworks for studying and assessing the development of computational thinking. *Paper presented at the Proceedings of the 2012 annual meeting of the American Educational Research Association*, Vancouver, Canada. Retrieved June 15, 2018, from http://scratched.gse.harvard.edu/ct/files/AERA2012.pdf

Büyüköztürk, Ş. (2018). *Sosyal Bilimler için Veri Analizi El Kitabı* (24. Baskı). Pegem Yayıncılık, Ankara.

Carnegie Mellon University. (1997). About Alice. Retrieved June 15, 2018, from http://www.alice.org/index.php?page=what_is_alice/what_is_alice

Choi, H. (2012). Learners' reflections on computer programming using Scratch: Korean primary pre-service teachers' perspective. *Paper presented at the 10th International Conference for Media in Education* 2012 (ICoME).

desJardins, M. (2015). Creating AP® CS principles: let many flowers bloom. *ACM Inroads,* 6(4), 60-66. Retrieved June 15, 2018, from http://doi.org/10.1145/2835852

Dreyfus, S. E. (1986). *Dynamic programming The Bellman Continuum* (pp. 13-70): World Scientific.

Du Boulay, B. (1986). Some difficulties of learning to program. *Journal of Educational Computing Research*, 2(1), 57-73. Retrieved June 15, 2018, from http://journals.sagepub.com/doi/pdf/10.2190/2193LFX-2199RRF-2167T2198-UVK2199

Fesakis, G., & Serafeim, K. (2009). Influence of the familiarization with scratch on future teachers' opinions and attitudes about programming and ICT in education. P*aper presented at the ACM SIGCSE Bulletin*. Retrieved June 15, 2018, from http://dl.acm.org/citation.cfm?id=1562957

Fessakis, G., Gouli, E., & Mavroudi, E. (2013). Problem solving by 5–6 years old kindergarten children in a computer programming environment: A case study. *Computers & Education, 63*, 87-97. Retrieved June 15, 2018, from http://www.sciencedirect.com/science/article/pii/S0360131512002813

Fowler, A., Fristoe, T., & MacLaurin, M. (2012). Kodu Game Lab: a programming environment. *The Computer Games Journal,* 1(1), 17-28. Retrieved June 15, 2018, from https://pdfs.semanticscholar.org/d998/d996a997e934bc952f996e279037c263781a279035f279037a275467a.pdf

Fraenkel, J., & Wallen, N. (2009). *The nature of qualitative research. How to design and evaluate research in education, seventh edition*. Boston: McGraw-Hill, 420.

Gouws, L. A., Bradshaw, K., & Wentworth, P. (2013). Computational thinking in educational activities: an evaluation of the educational game light-bot. *Paper presented at the Proceedings of the 18th ACM conference on Innovation and technology in computer science education*. Retrieved June 15, 2018, from http://dl.acm.org/citation.cfm?id=2466518

Grover, S., & Pea, R. (2013). Computational Thinking in K–12 A Review of the State of the Field. *Educational Researcher,* 42(1), 38-43. Retrieved June 15, 2018, from http://journals.sagepub.com/doi/abs/10.3102/0013189X12463051

Günüç, S., Odabaşı, H. F., ve Kuzu, A. (2013). 21. yüzyıl öğrenci özelliklerinin öğretmen adayları tarafından tanımlanması: Bir Twitter uygulaması. *Eğitimde Kuram ve Uygulama,* 9(4), 436-455.

Heintz, F., Mannila, L., & Färnqvist, T. (2016). A review of models for introducing computational thinking, computer science and computing in K-12 education. *Paper presented at the Frontiers in Education Conference (FIE), 2016 IEEE*. Retrieved June 15, 2018, from http://doi.org/10.1109/FIE.2016.7757410

Johnson, L., Adams Becker, S., Estrada, V., Freeman, A., Kampylis, P., Vuorikari, R., & Punie, Y. (2014). *Horizon Report Europe: 2014 Schools Edition.* Luxembourg: Publications Office of the European Union, & Austin, Texas: The New Media Consortium.

Kalelioğlu, F. & Gülbahar, Y. (2014). The effects of teaching programming via Scratch on problem solving skills: A discussion from learners' perspective. *Informatics in Education.* 13(1), 33-50.

Kay, K. & Greenhill, V. (2011). Twenty-first century students needs 21st century skills. In G. Wan & D. M. Gut (Eds.), *Bringing Schools into the 21st Century* (pp. 41–66). Dordrecht, Germany: Springer. Retrieved June 15, 2018, from https://doi.org/10.1007/978-94-007-0268-4

Kelleher, C., & Pausch, R. (2005). Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys (CSUR),* 37(2), 83-137. Retrieved June 15, 2018, from http://dl.acm.org/citation.cfm?id=1089734

King, F., Goodson, L., & Rohani, F. (2010). *Higher order thinking skills: Definition, teaching strategies, assessment.* Publication of the Educational Services Program, now known as the Center for Advancement of Learning and Assessment. Retrieved June 15, 2018, from www.cala.fsu.edu

Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., . . . Seppälä, O. (2004). A multi-national study of reading and tracing skills in novice programmers. *Paper presented at the ACM SIGCSE Bulletin*. Retrieved June 15, 2018, from https://doi.org/10.1145/1044550.1041673

Lye, S. Y., & Koh, J. H. L. (2014). Review on teaching and learning of computational thinking through programming: What is next for K-12? *Computers in Human Behavior,* 41, 51-61. Retrieved June 15, 2018, from http://www.sciencedirect.com/science/article/pii/S0747563214004634

Maloney, J., Resnick, M., Rusk, N., Silverman, B., & Eastmond, E. (2010). The scratch programming language and environment. *ACM Transactions on Computing Education (TOCE),* 10(4), 16. Retrieved June 15, 2018, from http://dl.acm.org/citation.cfm?id=1868363

Microsoft. (2017). About Small Basic. Retrieved June 15, 2018, from, http://smallbasic.com/about.aspx

Ministry of Education. (2012). 12-Year Compulsory Education Questions - Answers. Retrieved June 15, 2018, from http://www.meb.gov.tr/duyurular/duyurular2012/12Yil_Soru_Cevaplar.pdf

MIT, M. I. o. T. (2017a). About App Inventor. Retrieved June 15, 2018, from http://appinventor.mit.edu/explore/about-us.html

MIT, M. I. o. T. (2017b). About Scratch. Retrieved June 15, 2018, from https://scratch.mit.edu/about

Ozoran, D., Cagiltay, N., & Topalli, D. (2012). Using scratch in introduction to programming course for engineering students. *Paper presented at the 2nd International Engineering Education Conference (IEEC2012).*

Pears, A., Seidman, S., Malmi, L., Mannila, L., Adams, E., Bennedsen, J., . . . Paterson, J. (2007). A survey of literature on the teaching of introductory programming. *ACM SIGCSE Bulletin,* 39(4), 204-223.

Pierce, B. C. (2002). *Types and programming languages.* Retrieved June 15, 2018, from http://robotics.upenn.edu/~bcpierce/tapl/contents.pdf

Repenning, A., Basawapatna, A., & Escherle, N. (2016). Computational thinking tools. *Paper*

presented at the Visual Languages and Human-Centric Computing (VL/HCC), 2016 IEEE Symposium. Retrieved June 15, 2018, from http://ieeexplore.ieee.org/abstract/document/7739688/

Robins, A., Rountree, J., & Rountree, N. (2003). Learning and teaching programming: A review and discussion. *Computer science education,* 13(2), 137-172.

Saeli, M., Perrenet, J., Jochems, W. M., & Zwaneveld, B. (2011). Teaching programming in secondary school: a pedagogical content knowledge perspective. *Informatics in Education-An International Journal,* 10(1), 73-88.

Sağlam, M. (2014). The 4+ 4+ 4 in the Educational Experiences of the the Teachers Teaching the First Grade Students in Turkey: Yozgat City as an Example. *Journal of History School,* 7(18), 377-396.

Selby, C., & Woollard, J. (2013). Computational thinking: the developing definition. Retrieved June 15, 2018, from https://eprints.soton.ac.uk/356481/1/Selby_Woollard_bg_soton_eprints.pdf

Spohrer, J. C., & Soloway, E. (1989). Simulating student programmers. Ann Arbor, 1001, 48-109. Received from http://ijcai.org/Proceedings/189-101/Papers/087.pdf

Szlávi, P., & Zsakó, L. (2006). Programming versus application. *Paper presented at the International Conference on Informatics in Secondary Schools-Evolution and Perspectives*. Retrieved June 15, 2018, from http://doi.org/10.1007/11915355_5

Weintrop, D., Beheshti, E., Horn, M., Orton, K., Jona, K., Trouille, L., & Wilensky, U. (2016). Defining computational thinking for mathematics and science classrooms. *Journal of Science Education and Technology,* 25(1), 127-147.

Wikipedia. (2017). Programming. Retrieved June 15, 2018, from https://en.wikipedia.org/wiki/Computer_programming

Winslow, L. E. (1996). Programming pedagogy—a psychological overview. *ACM SIGCSE Bulletin,* 28(3), 17-22.

Xinogalos, S. (2012). An evaluation of knowledge transfer from microworld programming to conventional programming. *Journal of Educational Computing Research, 47*(3), 251-277.

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: 10.21585/ijcses.v2i3.33

Abstract

Evaluating the quality of students' programs 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. This study examined the quality of students' Scratch programs by 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, instructional 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), 2) students' negative attitude, high anxiety and low 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 (Kalelioğlu & Gülbahar, 2014; Moreno-León, Robles, & Román-González, 2016).

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 students develop a program by "dragging and snapping blocks." Researchers have proved that the BBP makes learning the programming vocabulary easier by 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).

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.

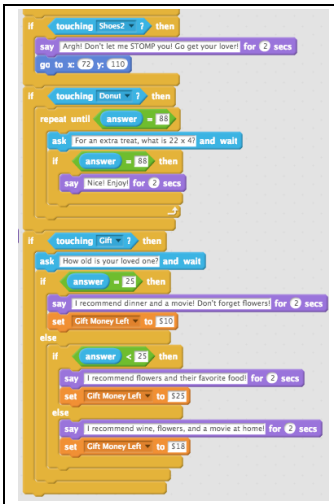
Q08 updated the "high score" using the if-block as Figure2-c. The code demonstrated how the student updated the value in consideration of the condition. (Please note that the forever-block inside the if-block was not necessary. We will discuss it later.)

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

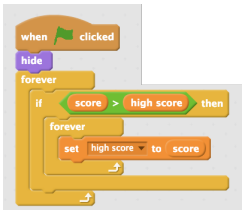
4.4.2 Students considered an alternative condition that was not necessary.

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.



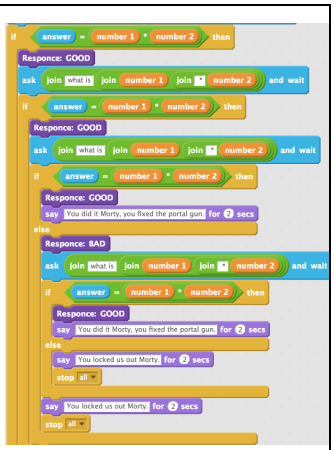
a



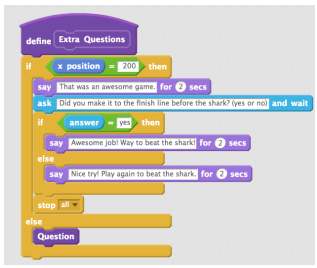
c

d

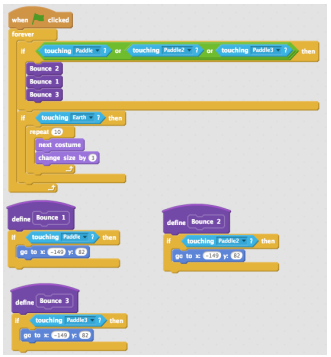
e



g



b



f

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.

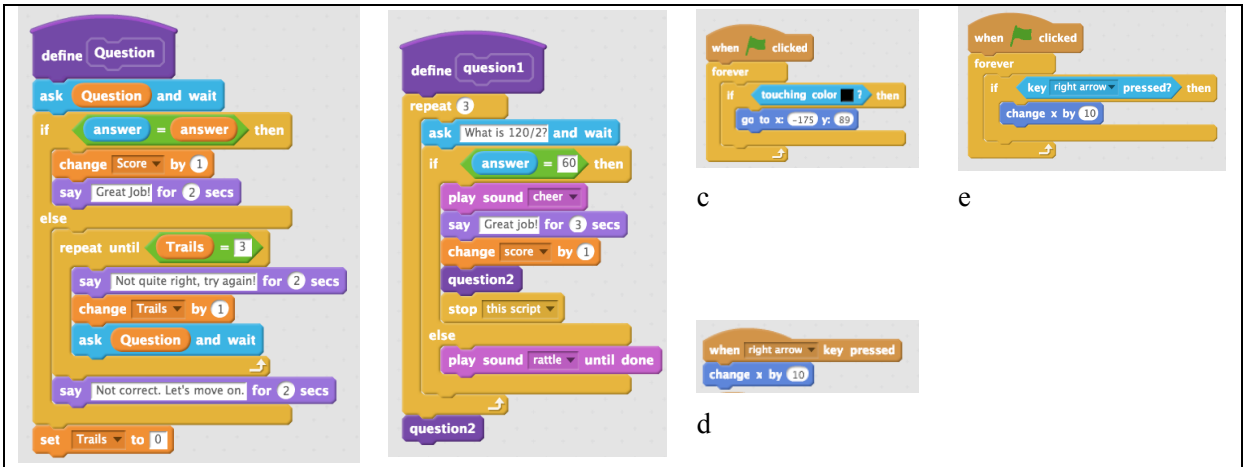


Figure 3. Codes representing the concepts of loops

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.

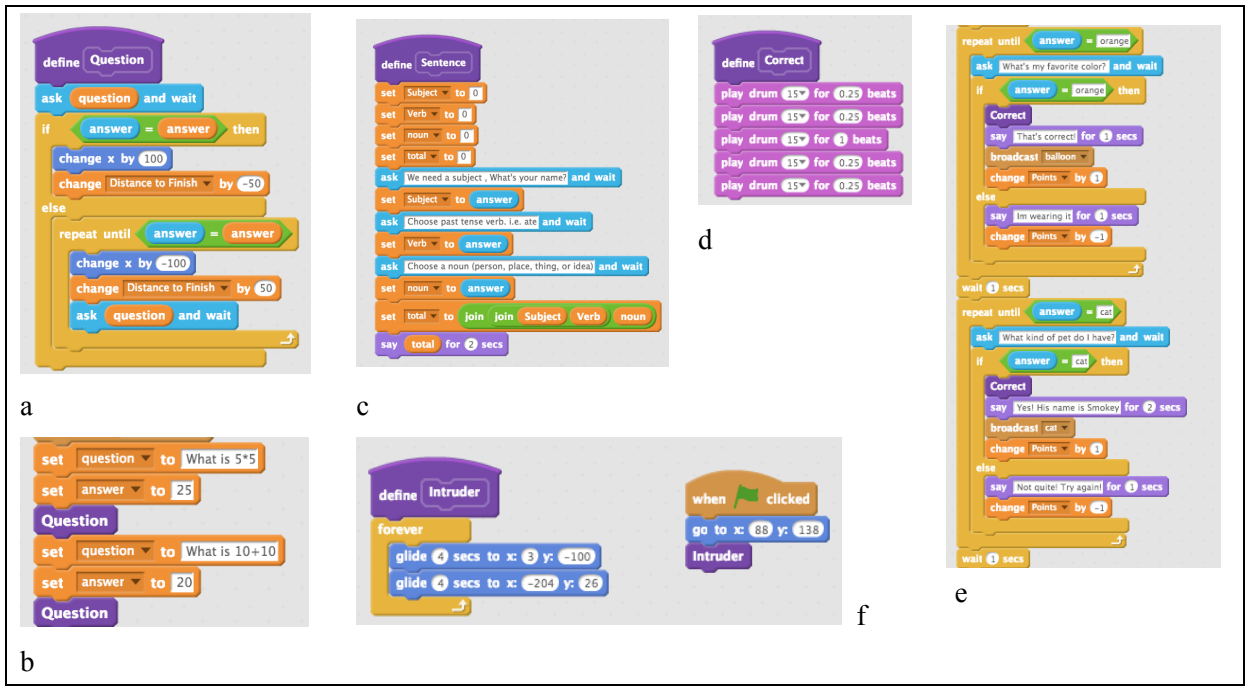


Figure 4. Codes representing the concepts of abstractions

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.

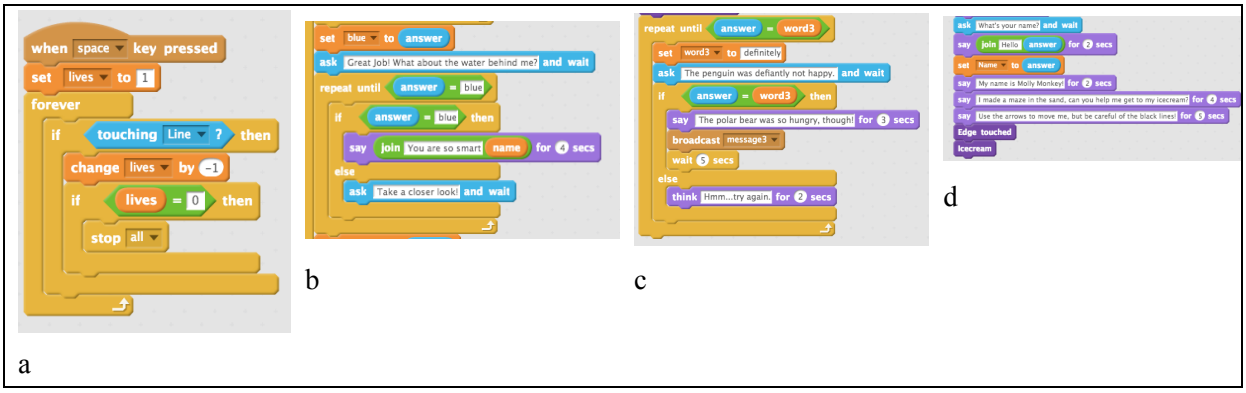


Figure 5. Codes representing the concepts of variables

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

IJCSES

Volume 2, Issue 3

August 2018

ISSN 2513-8359