Computer Science Education

ISSN: 0899-3408 (Print) 1744-5175 (Online) Journal homepage: www.tandfonline.com/journals/ncse20

Learning computer science concepts with Scratch

Orni Meerbaum-Salant, Michal Armoni & Mordechai (Moti) Ben-Ari

To cite this article: Orni Meerbaum-Salant, Michal Armoni & Mordechai (Moti) Ben-Ari (2013) Learning computer science concepts with Scratch, Computer Science Education, 23:3, 239-264, DOI: 10.1080/08993408.2013.832022

To link to this article: https://doi.org/10.1080/08993408.2013.832022

Published online: 02 Sep 2013.

Computer Science Education, 2013
Vol. 23, No. 3, 239–264, http://dx.doi.org/10.1080/08993408.2013.832022

Routledge
Taylor & Francis Group

Learning computer science concepts with Scratch

Orni Meerbaum-Salant, Michal Armoni* and Mordechai (Moti) Ben-Ari

Department of Science Teaching, Weizmann Institute of Science, Rehovot 76100, Israel

(*Received 14 April 2013; accepted 1 August 2013*)

Scratch is a visual programming environment that is widely used by young people. We investigated if Scratch can be used to teach *concepts* of computer science (CS). We developed learning materials for middle-school students that were designed according to the constructionist philosophy of Scratch and evaluated them in a few schools during two years. Tests were constructed based upon a novel combination of the revised Bloom taxonomy and the Structure of the Observed Learning Outcome taxonomy. These instruments were augmented with qualitative tools, such as observations and interviews. The results showed that students could successfully learn important concepts of CS, although there were problems with some concepts such as repeated execution, variables, and concurrency. We believe that these problems can be overcome by modifications to the teaching process that we suggest.

Keywords: Scratch; Bloom's taxonomy; SOLO taxonomy; middle schools; concurrency

1. Introduction

In an attempt to increase interest in computer science (CS), much effort has gone into developing tools and activities for young people, primarily middle-school students, although these tools and activities have been used both by younger children and as preliminary learning materials in high schools and universities. The activities include kinesthetic activities such as CS Unplugged (Bell, Witten, & Fellows, 2005) and magic shows (Curzon & McOwan, 2008), as well as visual programming environments such as Scratch (Resnick et al., 2009) and Alice (Dann, Cooper, & Pausch, 2009). Visual programming environments facilitate the development of software in a context that is fun and nonthreatening (Kelleher, Pausch, & Kiesler, 2007). The hope is that students will no longer feel anxiety and low self-esteem when faced with computers, so they will be more open to further study of

*Corresponding author. Email: michal.armoni@weizmann.ac.il
A preliminary version of this article was presented at the 6th International Computing Education Research Conference, Aarhus, Denmark, 2010.

CS. We chose to work with Scratch at the middle-school level because its simple two-dimensional world appeared sufficient for achieving the goal of learning CS concepts.

The Scratch website hosts a community with almost 1.5 million registered members and over 3 million projects have been uploaded. However, this says nothing about the concepts that students learn when they use Scratch. It is possible to concentrate on designing and modifying the costumes and sounds of the sprites, without paying any conscious attention to CS concepts. For example, only about 10% of the projects uploaded to the Scratch website use the conditional looping construct `repeat until <condition>` and only about 20% use variables.[1] From the perspective of CS educators, it is important to investigate if Scratch can actually facilitate the learning of concepts and to characterize how the concepts are learned, as a basis for preparing learning materials, and for training teachers and other mentors.

Our research was conducted in schools during normal school hours and the courses were taught by middle-school teachers using learning materials that we developed specifically for middle-school students. The research was carried out using both quantitative and qualitative tools. A pretest, an interim test, and a posttest were designed and given to the students. The tests were based upon a novel combination of the revised Bloom taxonomy (Anderson et al., 2001; Bloom, Mesia, & Krathwohl, 1964) and the Structure of the Observed Learning Outcome (SOLO) taxonomy (Biggs & Collis, 1982). A researcher observed most of the sessions, and interviewed the students and the teachers.

In Section 2 we review previous work and in Section 3 we describe the new taxonomy. Section 4 describes the learning materials. The research methodology, results, and discussion follow in Sections 5 through 7, and Section 8 concludes the paper.

2. Background

2.1. Learning of CS by young students

Interest in science and mathematics in general, and in CS in particular, declines as students progress in their studies (Carter, 2006). Lambert and Guiffre (2009) examined the effect of CS Unplugged on students' *interest* in CS, rather than their views on CS. They found that the students were more interested in CS after participating in CS Unplugged than before. Unfortunately, it has proved difficult to demonstrate that CS Unplugged actually achieves its wider long-term goals (Taub, Armoni, & Ben-Ari, 2012).

A survey of programming environments for young students can be found in (Kelleher & Pausch, 2005). We are particularly interested in visual programming environments because we believe that they are best able to foster learning of CS. Developing software is *hard fun* to use a phrase of Seymour

Papert: "I have no doubt that this kid called the work fun because it was hard rather than in spite of being hard" (Papert, n.d.), and thus is likely to facilitate learning.

Alice was found to improve college students' attitudes towards CS (Moskal, Lurie, & Cooper, 2004). Storytelling Alice is a variant of Alice that enables users to create stories with interesting characters; middle-school girls using the system displayed greater motivation and willingness to spend extra time on the computer when compared with those using generic Alice (Kelleher, Pausch, & Kiesler, 2007).

2.2. Scratch

Scratch (http://scratch.mit.edu), created by the Lifelong Kindergarten Group at the MIT Media Laboratory, is a media-rich system for novice programmers. For an overview of Scratch, see Resnick et al. (2009). Programs in Scratch are composed of *scripts* which control *sprites* displayed on a stage. Scripts are created by dragging and dropping blocks that represent program components, such as expressions, conditions, statements, and variables. A sprite may have multiple scripts and they are all executed concurrently. The environment eliminates syntax errors and gives immediate visual feedback through the behavior of the sprites. Scratch is augmented by a social computing network for sharing projects.

Scratch is frequently used in extracurricular activities. For example, Maloney, Peppler, Kafai, Resnick, and Rusk (2008) reported on an experiment where Scratch was used by students in an after-school clubhouse. These students were self-selected and self-paced, receiving no formal instruction. By analyzing the students' projects, they found that the majority of the projects that actually constructed executable scripts used both sequential and concurrent execution. They found that "*without realizing it*, most Scratch users make use of multiple threads" (p. 368, our emphasis); in the absence of conscious knowledge of what they are doing, we cannot conclude that the students learned and understood this important concept.

Wilson and Moffat (2010) examined the use of Scratch for introducing eight-year-old students to programming. These children studied Scratch for eight weeks, one-hour per week, during school hours, as part of their lessons in Information and Communication Technologies. The students enjoyed the lessons and were more enthusiastic than in other lessons. The authors only reported on two mean grades (52 and 64) of 12 students in two quizzes (which were not given in the paper). Given the limited scope of this research, we cannot draw conclusions on the knowledge gained by the students.

Scratch is also used as a gentle introduction to programming at the tertiary level, because it appears to facilitate the transition to more traditional environments (Wolz, Maloney, & Pulimood, 2008). According to Malan and

Leitner (2007), students found Scratch exciting and it successfully familiarized inexperienced students with the fundamentals of programming. However, this research was based on surveys alone. In a panel session, Wolz, Leitner, Malan, and Maloney (2009) reported that after initially learning Scratch, the students' transition to Java or C appeared to be easier.

2.3. Learning concepts

We were interested in determining whether middle-school students are capable of learning *concepts* of CS rather than just programming skills. A concept is a general notion, an abstract description of a set of objects that grasps the permanent aspects of all the objects in this set and their essence (Schwill, 2004). Yet, it is not as abstract as an idea, which postulates a meaning of objects and their underlying concepts. For example, a stack, and even a data structure are both concepts (on different level of abstraction), while data abstraction is an idea.

The difficulties that students have with CS concepts are well documented, both in general (Robins, Rountree, & Rountree, 2003), and specifically for each concept: variables (Kuittinen & Sajaniemi, 2004; Samurçay, 1989), loops (Dancik & Kumar, 2003), Boolean conditions and control structures (Almstrum, 1999; Herman, Loui, & Zilles, 2010), message passing (Paul, Kouril, & Berman, 2006), and concurrency (Ben-David Kolikant, 2001, 2004). We focused on these (and similar) concepts that are simultaneously fundamental in CS and central to developing programs in Scratch.

2.4. Taxonomies for assessment of learning

Two widely used taxonomies for assessment of learning are the *Bloom taxonomy* and the *SOLO taxonomy*. The Bloom taxonomy exists in two forms: the original form (Bloom, Mesia, & Krathwohl, 1964) and a revised form (Anderson et al., 2001). Both taxonomies have been used in CS education research: for example, the Bloom taxonomy was used in Whalley et al. (2006) and the SOLO taxonomy (Biggs & Collis, 1982) in Lister, Simon, Thompson, Whalley, and Prasad (2006). The taxonomies define hierarchical categories of learning. An assessment instrument that checks a student's achievement at each category enables one to state that the student has achieved a certain level of learning but no higher. This can be used by a teacher to tailor individual instruction and it can be used by a researcher to assess the effectiveness of a pedagogical approach or tool.

The revised Bloom taxonomy has two dimensions (Anderson et al., 2001): a cognitive dimension with six categories: *remembering, understanding, applying, analyzing, evaluating, and creating*, and a knowledge dimension

with four categories: *factual knowledge, conceptual knowledge, procedural knowledge, and metacognitive knowledge*. The cognitive categories are ordered from simple to complex and from concrete to abstract, and each category is assumed to include lower ones in an inclusive hierarchy. In the revised taxonomy, the requirement of a strict hierarchy was somewhat relaxed to allow some overlapping between categories (Krathwohl, 2002), but the general structure is still hierarchical.

The interpretation of the revised Bloom taxonomy for CS tasks is not straightforward (Fuller et al., 2007; Thompson, Luxton-Reilly, Whalley, Hu, & Robbins, 2008), and research shows that experts find it difficult to agree on an interpretation (Johnson & Fuller, 2006). Lister and Leaney (2003) claim that writing code can belong in different cognitive levels – understanding, applying, or creating – depending on the magnitude of code, and, similarly, tracking code (following the execution step-by-step) can belong to remembering, understanding, or analyzing, again depending on the magnitude of the code.[2] Johnson and Fuller (2006) argue that applying is the main goal of CS practice and suggest adding a level called *higher applying*. Thompson et al. (2008) interpret tracking code as understanding, but they classify writing simple bits of code under understanding as well. Fuller et al. (2007) suggest a new taxonomy appropriate for CS education; it is based on the Bloom taxonomy, but the cognitive dimension is restructured as a two-dimensional matrix differentiating between tracking code and writing code skills.

The SOLO taxonomy has five ordered categories:

- *Prestructural*: Mentioning or using unconnected and unorganized bits of information which make no sense;
- *Unistructural*: A local perspective where mainly one item or aspect is used or emphasized. Others are missing, and no significant connections are made;
- *Multistructural*: A multi-point perspective, where several relevant items or aspects are used or acknowledged, but significant connections are missing and a whole picture is not yet formed;
- *Relational*: A holistic perspective in which meta-connections are grasped. The significance of parts with respect to the whole is demonstrated and appreciated;
- *Extended abstract*: Generalization and transfer so that the context is seen as one instance of a general case.

Whalley et al. (2006) used both taxonomies to analyze the same data, but this induced two independent analyses even though the taxonomies are not independent. Thompson et al. (2008) noted that a category of the Bloom taxonomy can sometimes reflect a few SOLO categories.

3. A new taxonomy

Given the difficulties in categorizing programming activities within the existing taxonomies, we developed a new one by combining the revised Bloom and SOLO taxonomies.

While the Bloom taxonomy was appealing as a potential design and analysis tool, we felt that in our context it would be too general and not informative enough. For example, creating is considered to be much more complex than understanding, but can we really say that creating a simple project (e.g. a project containing a single script that moves a sprite from one point to another) is cognitively more complex than fully understanding the concept of concurrency? We wanted to work with a strictly hierarchal taxonomy, enabling us to monitor students' progress, but one that matched the context of the study and its objectives.[3] Intersecting the cognitive dimension with the knowledge dimension, as done in the revised Bloom taxonomy (Anderson et al., 2001) enriched the original taxonomy, but we still felt that this enrichment did not cope with the difficulty just described. We felt that putting tasks of the same kind into different categories because of differences in the magnitude of the code (as done by Lister & Leaney, 2003) is not the proper solution, and that the issue of magnitude should be addressed explicitly.

The SOLO taxonomy seemed to grasp another dimension – the spectrum between a local perspective (look at one tree) and a holistic perspective (seeing the whole forest). This categorization seemed to answer the above problem. For example, a full understanding of the concept of concurrency requires a holistic point of view, because it concerns not a single construct but rather the relationships and interactions of multiple constructs (sprites and scripts). This contrasts with creating a simple project that might require only local knowledge of a single short script that is executed by one sprite. The SOLO taxonomy also captures the magnitude of code issue, since tracking or producing small programs usually requires a relatively local perspective, while more complex code requires multistructural and relational abilities. The SOLO taxonomy is indeed very useful when analyzing a single activity such as analyzing the levels at which students track the execution of programs or their levels of program creation. However, using the SOLO taxonomy for various types of activities such as tracking and program creation, simultaneously, is not straightforward.

We chose to create a taxonomy that is a combination of the two. From the SOLO taxonomy, we chose to focus on the three intermediate categories of unistructural, multistructural, and relational. From the Bloom taxonomy, we chose to focus on understanding, applying, and creating. Our interpretation of these three Bloom categories in the context of CS education is as follows (using terms taken from the cognitive dimension of the revised taxonomy (Anderson et al., 2001)):

- *Understanding*: The ability to summarize, explain, exemplify, classify, and compare CS concepts, including programming constructs;
- *Applying*: The ability to execute programs or algorithms, to track them, and to recognize their goals;
- *Creating*: The ability to plan and produce programs or algorithms.

We combined these two sets of categories to form a taxonomy with three super-categories – unistructural, multistructural, and relational – each containing three sub-categories – understanding, applying, and creating. This yielded a nine-level taxonomy with the highest level being relational creating, and the lowest level being unistructural understanding.

The category unistructural creating, for example, describes the ability to create very small scripts doing one thing, or adding an instruction with local effect to an existing script. The category relational understanding describes the ability to fully understand a complex concept (such as concurrency) and to coherently explain its various facets. The category multistructural applying describes the ability to track a program and grasp its various parts but not the whole entity they form.

Our new taxonomy is hierarchical and the hierarchy implies that it is more difficult for a student to perform at a higher level than a lower level. We claim that performance at higher levels of the taxonomy does imply a deeper comprehension of a concept than the superficial familiarity implied by lower levels. Furthermore, we expect that there will be an inverse relationship between the level of a task and the mean grade achieved by the students on that task.

In Section 5, we give examples how the new taxonomy is used to categorize questions.

4. The learning materials

When a tool like Scratch is used in a school setting, it is important that high-quality learning materials be available, so that teachers can follow a specific syllabus and not be required to develop every lesson by themselves.[4] This is especially important for middle-school teachers who tend to have a less-advanced academic background in the subject matter.

By *learning materials* one normally means a textbook, but a routine textbook would not be compatible with the constructionist educational philosophy upon which Scratch is based (Papert & Harel, 1991). The second and third authors developed a textbook[5] that explains everything we want the students to learn in full gory detail, but we do not expect most students to read it routinely; instead, it is intended primarily as a guide for teachers, parents, and mentors, although independent learners will be able to use it for self-study.

The textbook is structured according to the following principles:

- Each chapter teaches a specific CS *concept* rather than a Scratch feature and the concepts are arranged in an order that we believe is best for learning about CS. In particular, we introduce concurrency very early, postponing classical concepts of programming (such as variables).
- Programming constructs are introduced *as needed*. For example, we do not introduce and explain all possible looping constructs in a single section, but rather introduce each one when it is needed. Of course the problems we pose are carefully selected so that their solutions just happen to need the construct we want to teach at that moment.
- The presentation is *project based*. Each chapter is based upon a specific context such as animating dancers or a Pac-Man game. For each project, a sequence of tasks is presented to the learner: make the Pac-Man sprite open and close its mouth, make the sprite move, enable the user to control the direction in which the sprite moves, and so on, until a reasonably complete game has been created. By constructing programs to implement the tasks one after another, the learner ends up with a working software artifact while being taught concepts and programming constructs along the way. The full Scratch source code for all 150 examples and exercises is available for download.
- We *de-emphasize aspects of appearance* such as the costumes, sounds, and visual effects. Instruction on these topics is included as optional tasks at the end of each chapter. This is an attempt to counteract the natural tendency of young people to engage in non-CS activities like drawing and playing music; for example, 21% of the students in a clubhouse used Scratch only for media manipulation (Maloney et al., 2008).

5. Research methodology

5.1. Research questions

We posed the following questions at the beginning of our research project:

- Is Scratch effective for teaching and learning CS?
- How does the effectiveness of learning vary for different CS concepts?

5.2. Research population

The research population consisted of middle-school students with no previous exposure to CS. During the first year (2009–2010), there were two ninth-grade classes, one with 18 students and one with 28 students.

During the second year (2010–2011), there were five ninth-grade classes (number of students: 34, 24, 29, 30, 18) and one eighth-grade class (23 students). Scratch was taught for one two-hour period per week for about 20 weeks.

Two different teachers taught the two classes in the first year and continued to the second year; one of them taught two classes that year. Three new teachers participated in the second year, each teaching one class. The teachers' backgrounds varied in terms of their experience teaching CS. For example, one of the teachers in the first year taught middle-school mathematics, while the other had 15 years of experience teaching CS in high schools. None of the teachers had experience with Scratch. During the summer between the two years, a course on teaching CS with Scratch was offered. Four of the five teachers participated in the course, two of the three new teachers in the second year and the two teachers who continued from the previous year.

The teachers were given our textbook together with its Scratch projects. We composed quizzes that were used by the teachers for evaluation in class and by us as research instruments. The researchers did not intervene during the teaching of the classes, but we were available for technical and pedagogical support when requested by the teachers.

Given the small number of teachers, we did not investigate the effect of the teachers' backgrounds on the learning outcomes.

5.3. Data collection

We used a mixed-method research methodology that combined qualitative and quantitative techniques (Johnson & Onwuegbuzie, 2004). A variety of data was collected.

There were three tests: a pretest, an interim test, and a posttest. The pretest was given during the first lesson before the teachers started teaching. The interim test was given during the sixth lesson after the first five chapters had been studied. The posttest was given in the tenth lesson after two more chapters had been studied. Given that the subjects were novices with no background in CS, the pretest was at an abstract algorithmic level. Its purpose was to verify that they had no meaningful knowledge of CS and thus establish a baseline. The interim test and posttest can be considered as a single test whose purpose is to measure learning of CS concepts. Two tests were used so that concepts could be tested close to when they were studied, and also to take into account the limited amount of time customary for middle-school tests.

Semi-open interviews were held with five teachers and 10 students.

The first author conducted non-participant observations during most of the class sessions.

Reflective interviews were conducted with the teachers whose classes were observed.

The students' work, including solutions to exams, was collected. In addition, Scratch projects that students submitted for presentation at a public Scratch day held at our institution were collected.

A focus group was held to investigate students' perceptions regarding their first exposure to the basic ideas and concepts of CS.

5.4. The use of the new taxonomy

The tests (interim test and posttest) were designed to maximize the coverage of the categories of the taxonomy, while at the same time limiting the extent of the test to what is appropriate for middle-school students.[6] The design of the tests was guided by the combined taxonomy, with each question matching a cognitive category in the sense that a full solution to it demonstrates that the student has learned at the level of the category. Table 1 shows the categories that were included in each of the tests, where there was sometimes more than one question per category. The authors categorized the questions independently according to the combined taxonomy and then negotiated a consensus in cases of disagreement.

The following CS concepts were included in the tests: initialization, repeated execution (unbounded, bounded, and conditional), conditional execution (only in the second year), communication by message passing, variables, event handling (only in the second year), and concurrency. We divided the concept of concurrency in two concepts: Type I concurrency occurs when several sprites are executing scripts simultaneously, for example, sprites representing two dancers. Type II concurrency occurs when a single sprite executes more than one script simultaneously, for example, a Pac-Man sprite moving through a maze while opening and closing its mouth.

Here are examples of questions corresponding to some of the categories of the taxonomy:

5.4.1. Unistructural

5.4.1.1. Applying. Referring to a script whose next to the last instruction is point in direction:

Table 1. Categories included in each test.

	Unistructural			Multistructural			Relational		
	U	A	C	U	A	C	U	A	C
Interim test		X	X		X				
Posttest				X	X	X		X	X

Notes: U = Understanding, A = applying, C = creating.

In which direction is the cat facing at the end of the script?

(a) −90, (b) 0, (c) 90, (d) 180

5.4.2. Multistructural

5.4.2.1. Understanding. What is the difference between the instruction `say [hello]` and the instruction `broadcast [hello]`?

5.4.2.2. Creating. Add to the following scripts an instruction or instructions that will cause the scenario where the sprites change places to repeat itself five times.

5.4.3. Relational

5.4.3.1. Applying. Describe the behavior of the animals and the ball during the execution of all the scripts (after clicking on the green flag).

5.4.3.2. Creating. Construct an animation with two sprites. The sprites will be placed in two corners of the stage facing the center of the stage. Pick one sprite whose task will be to broadcast the message *switch* to the other sprite. After the message is received the two sprites will change their places using the instruction Courier new `secs to` *x*:(…) *y*:(…). During the process of changing places, the sprites will say something to each other when they meet.

5.5. Data analysis

The tests were graded like normal school tests except in cases where a question referred to more than one concept. In those cases, separate grades were given for each concept. A correct answer is interpreted as demonstrating that the student performed at the level of the category of the question. When a high percentage of students achieve good grades on a question, we can conclude that it is possible to learn the given CS concept at the level of the question using the Scratch environment. A qualitative analysis of the observations and interviews is used to reinforce the conclusions from the quantitative data.

6. Findings

The analysis during the first year focused on the cognitive levels without distinguishing between the levels achieved for each individual concept.[7] The only exceptions were the two concurrency concepts, where the analysis took the concept into account. Since we found that this more refined analysis gave us important information, we decided that during the second year we would perform the analysis both by level and by concept for all concepts.

The pretest from both years confirmed that the students started their studies with hardly any knowledge of CS concepts. The findings will thus focus on the levels of knowledge achieved in the interim test and the posttest.

6.1. Achievements according to cognitive levels during the first year

Analyzing the data from the first year, we grouped the questions in each test according to the different cognitive categories, and calculated the mean grades of students' solutions in each of the groups (Table 2). Looking at the interim test and the posttest, students' grades show the expected decrease from high grades for lower cognitive categories (78, 81 for unistructural applying and creating) to lower grades for higher cognitive categories (33 for relational creating). The anomaly is the high grade for relational applying (73), which will be discussed in Section 7.

6.2. Findings according to CS concepts

In the first year, one of the questions in the posttest asked the students to define several concepts. The data obtained from this question enabled us to deduce information regarding students' multistructural understanding for each of these concepts. In the second year, we graded each of the concepts that appeared in each of the questions. (As noted above, since a question could require knowledge of multiple concepts, each concept was graded separately.) Then we grouped the questions by cognitive levels. For each pair (concept, cognitive level), we calculated the mean grade. Table 3 presents the second-year grades for all concepts (I = interim test, P = posttest). In the second year, 108 students (out of 158) took the interim test and the posttest: 25, 7, 13, 26, and 18 students from the ninth-grade classes and 19 students from the eighth-grade class.

We now present the results for each of the concepts that were checked. The quantitative results are reinforced by qualitative findings as appropriate.

6.3. Initialization

In the context of Scratch, the concept of initialization is not solely connected with variables; one usually has to initialize the position and direction of

Table 2. Grades for the first year by cognitive category.

	Unistructural			Multistructural			Relational		
	U	A	C	U	A	C	U	A	C
Interim test, $N = 43$		78	81		55				
Posttest, $N = 40$				55	46	51		73	33

Notes: U = Understanding, A = applying, C = creating.

sprites, and other attributes, such as a sprite's costume. Looking at the data from the first year, the mean grade of 18 for the definition question shows that few students could give a satisfactory *definition* for this concept in the posttest. Even though they could not define initialization, we found that their Scratch programs given as answers to questions in the posttest included adequate initialization instructions. The same was observed in the projects they developed. Therefore, we see that students were able to perform at the level of creation even though their multistructural understanding (by defining) was not satisfactory.

The quantitative findings from the second year (Table 3) give more information. In the interim test, the concept of initialization was measured in two cognitive levels, unistructural applying, and multistructural applying. Comparing students' achievements on these two levels regarding the concept of initialization, we can see that the performance decreases as the cognitive level increases. In the posttest, this concept was measured in four cognitive levels. For the three highest levels – multistructural applying, relational applying, and relational creating – we can again see a decrease in performance as the level increases. However, for the lowest level of the four, multistructural understanding, the grade was also the lowest. In this case too, the grade for this lower level of multistructural understanding was given according to the question in which the students were asked to define the concept. Thus, this anomaly is actually consistent with the findings obtained in the first year, since it seems that the students knew how to use this concept but had difficulties in defining it.

Looking at students' explanations from both years, we see quite a few definitions that indicate a confusion of initialization and the action that takes place when the green flag is clicked: "Initialization is to reactivate," "… to restart," "… to start over all actions," "… to start over the whole process," and "… to press the green flag."[8] Although this could indicate that these students did not understand the concept of initialization, it is possible that the result arose from a simple linguistic confusion. Since the teachers usually presented concepts without explicitly naming and defining them, the students were perhaps aware of the concept of initialization but were not fully aware of its name. In this case, they may have confused the words "start" and "initialization," which are very similar in Hebrew, and this can explain the anomaly mentioned above of a low grade on multistructural understanding.

6.4. Repeated execution

Repeated execution (bounded and conditional) was taught in Chapter 5. Our findings indicate that the students learned the concepts to some degree. In the first year, the mean grade of 58 showed that many students correctly defined (multistructural understanding) bounded repeated execution, and this

Table 3. Grades by cognitive concepts (second year).

$N = 108$	Unistructural applying		Unistructural creating		Multistructural understanding		Multistructural applying		Multistructural creating		Relational applying		Relational creating	
	I	P	I	P	I	P	I	P	I	P	I	P	I	P
Initialization	94					49	65	89				88		68
Bounded repeated execution						63	44							43
Conditional repeated execution			36			62	40							
Unbounded repeated execution			71							76				
Variables						69				26				
Conditional execution						71						82		
Communication: message passing						85						83		79
Event handling	89		63				90			26		82		57
Concurrency I						64	41	92				88		89
Concurrency II						10								10

increased to 75 for defining conditional repeated execution. The findings from the second year are similar: 63 for bounded repeated execution and 62 for conditional repeated execution (Table 3, the posttest column of multistructural understanding).

However, students had difficulties in multistructural applying for bounded (44) and conditional (40) repeated execution, and for relational creating of bounded (43) repeated execution. They did much better for unbounded repeated execution: 71 for unistructural creating and 76 for multistructural creating.

Looking at the qualitative data, we saw that sometimes students refrained from using (creating) repeated execution, and instead used other control structures, such as conditional execution (if) and unbounded repeated execution (taught as early as Chapter 3), usually incorrectly. Sometimes, they achieved the affect of conditional repeated execution by combining conditional execution and unbounded repeated execution, although this could suffer from race conditions.

In another publication (Meerbaum-Salant, Armoni, & Ben-Ari, 2011), we attributed this phenomenon to a habit of programming that was observed for these students, and that we called *extremely fine-grained programming*. As discussed in that paper, this habit was probably fostered by the Scratch environment, and one of its side effects was the decrease in the use of certain control structures, such as bounded or conditional repeated execution, thus interfering with learning the concepts underlying these structures.

6.5. Variables

Unlike many other CS concepts which appear in an unusual guise in Scratch, the support for variables in Scratch is quite similar to that in regular programming languages. One has to declare a variable (as either global or local to a sprite) and then assign values using blocks such as set or change. Values are read using reporter blocks that hold variable names. In our learning materials, this concept is introduced relatively late (Chapter 6) and thus was included only in the posttest.

In the first year, the grade of 8 showed that few students could give a correct definition for this concept (multistructural understanding). Things were much better in the second year (Table 3), with a mean grade of 69 for this cognitive level. This will be discussed in Section 7. The mean grade for multistructural creating level was rather low (26).

The qualitative findings uncovered some interesting phenomena. Many students identified the concept of a variable with the actions of updating it, initializing it, or creating it. This shows that the students have transformed the variable object into processes on the object. Some of the students seemed to interpret the concept of a variable in the restricted context of counting: “it is an action that counts in some way things that happen to a

sprite," "it is a counter that keeps changing by what one defines for it," "you create a variable to increase or decrease points, to add or decrease time, etc." Others seemed to interpret a variable in the narrow context of its visual representation on the stage: "something that appears in the upper corner of the computer," and others interpreted it in the familiar algebraic context. These restrictions by the students imply a decrease in their levels of the SOLO taxonomy.

6.6. Conditional execution

In our textbook, the concept of conditional execution is taught relatively late (Chapter 7). For that reason, it was not covered in the posttest of the first year.

The students' grades with regard to conditional execution were quite high (Table 3): 71 for multistructural understanding and 82 for relational applying.

Although this concept was taught late, we observed students using it spontaneously, even before being introduced to it by the teachers. Our qualitative findings show that they could also define the concept. A few of the students defined the concept by giving the Scratch instructions that implement it, while a very few students confused this concept with other, similar, conditional concepts such as conditional waiting. However, most of the students gave a satisfactory definition, referring to the condition and to the instructions which it controls. Here, too, the few observed restrictions show a decrease in levels of the SOLO taxonomy.

6.7. Communication by message passing

Message passing is a powerful tool in Scratch for communication between sprites and for synchronization. In the first year, the mean grade was 63 for defining the concept (multistructural understanding). In the second year (Table 3), we gathered information regarding two more cognitive levels. As expected, this performance decreases as the cognitive levels increase, but even on the highest cognitive level of relational creating, the grade is 79.

The concept of message passing is quite complex, combining concurrency, synchronization, and the asymmetric roles of the sender and receiver. Our qualitative findings indicate that some of the students grasped this concept at the multistructural understanding level only partially. For example, some students focus on the asymmetric roles, on one sprite managing other sprites: "to tell him what he needs to do," "you can tell other sprites that if they receive this message they should do something," "one sprite broadcasts a message, and every sprite that receives it should execute the instructions in the script." However, many students provided satisfactory definitions, capturing the essence of this concept, such as "we need to broadcast and receive messages when we want two different sprites to do something at the same

time," "to pass an order that would help the other sprite to function on the right time." Both these quotes refer to the asymmetric roles, as well as synchronization and concurrency.

6.8. Event handling

Event handling is a central concept in Scratch that appears in many constructs: message passing, clicking on a sprite, pressing a key, one sprite touching another, colors of sprites touching, and mouse clicks. Our findings indicate very good performance on event handling at the level of multistructural applying (90) and relational applying (82). The grade for the level of multistructural creating is low (26).

6.9. Concurrency

Scratch encourages the use of concurrency from the very beginning with multiple sprites and multiple scripts for each sprite. Concurrency appears in the learning materials as early as Chapter 2, but in an informal manner. Recall that we divided this concept in two: Type I concurrency occurs when several sprites are executing scripts simultaneously, while Type II concurrency occurs when a single sprite executes more than one script simultaneously. Type I concurrency should be simpler because it is associated with visually distinct sprites, while Type II concurrency is a more high-level concept that we taught later (Chapter 3).

According to our findings, Type I concurrency seems to be much more intuitive for students and easier to grasp. Students had problems with Type II concurrency when they solved the textbook assignments. We observed students who programmed instructions in a sequential order instead of concurrently for a single sprite. In addition, some students neglected to include the instruction `when green flag clicked` that activates execution at the beginning of all scripts; instead, they included it only at the beginning of one of the scripts.

In the first year, four questions dealt with concurrency concepts in the posttest, each testing a different cognitive category (Table 4). We can see that students' achievements in the two questions involving Type II concurrency were low, while their achievements in the other two questions were

Table 4. Grades for concurrency in the posttest of the first year.

Question	Cognitive category	Concurrency type	Grade
1	Multistructural understanding	I, II	8
2	Multistructural applying	I	38
3	Relational applying	I	63
4	Relational creating	I, II	3

higher. Comparing the results for the second and third questions led to a puzzling finding. The mean grade of 63 was achieved for the third question – categorized at the relational applying level – while the mean grade for second question – categorized at the multistructural applying level – was only 38. We will discuss this further in Section 7.

In the posttest, when asked to give a definition (multistructural understanding) of concurrency (Question 1, Table 4), the mean grade of 8 referred to both types of concurrency. The percentages of students who defined each type of concurrency were: 23% of the definitions were limited to Type I concurrency, 43% were limited to Type II concurrency, and 28% did not give any definition. Even those students whose definition reflected Type II concurrency did not succeed in writing a program (relational creating) using Type II concurrency (Question 4, Table 4); only one student answered this question correctly.

The findings of the second year (Table 3) depict a similar picture. The students demonstrated satisfactory performance when dealing with Type I concurrency, but a very low performance when dealing with Type II concurrency.

Looking at the qualitative data, we can see a phenomenon that is similar to that found for the concept of communication by message passing. The concept of concurrency is also a complex concept, combining Type I concurrency, Type II concurrency, synchronization, and a concrete instructions like those for message passing used to implement concurrency. It was not unexpected that we found some students who identified only some of these elements. The following quotes demonstrate the limited perceptions of concurrency:

- Type I concurrency: "Actions of different sprites that happen on the same time";
- Synchronization: "Concurrent execution is when you want to execute concurrently, that is, two sprites together, and you can do this by broadcasting a message";
- Concurrency identified with message passing: The previous quote and: "Like when you give a message to two sprites, so they both run concurrently";
- Concurrency identified with events: "When we want to have a concurrent script, when we press on something it will run, when we click the green flag, when a message is received, or when we press on the space bar";
- Concurrency identified with a geometric context: One student simply drew a parallelogram, probably because the word in Hebrew is almost the same as the word for concurrency.

6.10. Affective aspects

We found qualitative evidence of the positive effect that the course had on the students' motivation and interest in the field. The students reported in the interviews that Scratch changed their perception on CS, influencing their decision to take it at the high-school level:

- At the beginning I did not want to study computer science; I thought that I am not good at it. Last year I had a bad experience with learning Excel and I was really bad in that. So I realized that computer science is not for me. Exposure to Scratch has changed my thoughts, and I decided to extend my studies and to take an extra five units in an external program where I could develop projects by myself.
- I decided to study computer science in high school …. Scratch just gave me a little taste of what's going to happen next year and it intrigued me more and more. I'm really looking forward to keep learning more and more and looking forward to high school.
- The exposure to Scratch opened me to a new world and made me like the field.
- From the beginning I wanted to choose computer science as a main subject but the exposure to Scratch caused me to be sure in my decision.
- I enjoyed learning Scratch and especially programming computer games …. I enjoyed very much learning Scratch, especially I wanted to change animations and I really wanted to learn more and more.

One of the teachers reported in the interview that following the Scratch course in middle school, they opened two high-school CS classes instead of one, as in previous years. This led us to continue the research by investigating students who learned Scratch as they continue their CS studies in high school. We will report on the results in a subsequent article.

7. Discussion

The findings showed that a meaningful learning process of many (but not all) CS concepts occurred as a result of the courses in Scratch with the students performing at relatively high cognitive levels. Why is there such a difference in the learning of different CS concepts? Our first explanation is in terms of the mapping between concepts and constructs in the Scratch programming language. Then, we present a different approach in terms of the patterns of perception that we observed; these patterns will then be interpreted within the theoretical framework of reducing levels of abstraction.

7.1. Concepts vs. constructs

The performance level was not always high, especially for the concepts of variables, repeated execution and concurrency. These concepts are more abstract than other concepts that we investigated. A concept like initialization is mapped fairly straightforwardly into a simple sequence of Scratch instructions: one block initializes the position of a sprite, another block its direction, and a third its costume. One can successfully work with initialization at the unistructural levels. On the other hand, the above three concepts are implemented by instructions with a complex structure (repeated execution) or several instructions with complex inter-relationships (concurrency and variables). Therefore, successfully employing these concepts – even to a minimal extent – necessitates at least multistructural and even relational levels.

7.2. Patterns of perception

As demonstrated in Section 6, for most of the concepts we found four *patterns of perception*:

- *Flattening*: Identifying a concept with a subset of its aspects, for example, identifying concurrency with synchronization.
- *Specializing*: A concept is perceived as a special case of itself, for example, variables perceived as counters.
- *Concretizing*: Concepts are grasped as specific instructions or specific representations, for example, variables are grasped as a visual representation on the stage, or concurrency is grasped as the instructions for broadcasting and receiving messages. To some extent we saw concretizing for all concepts, since most students chose to explain or define a concept by giving a Scratch instruction that implements it.
- *Corresponding*: A concept is perceived in terms of a corresponding cognate in some other field like mathematics, for example, program variables were seen as corresponding to variables in algebra and concurrent execution corresponded to the geometric figure of the parallelogram.

7.3. Reducing levels of abstraction

Given the young age of the students, these patterns of perception might arise simply from the linguistic difficulties that students had in constructing explanations. However, these patterns can also be considered as manifestations of the reduction of the abstraction level of new concepts (Hazzan, 2003). According to Hazzan, when dealing with newly introduced concepts students attempt to reduce the level of abstraction in one of three ways:

(1) Making a new concept more concrete by making it more familiar. Thus, if the students are already familiar with another concept that resembles the new concept to some extent, they tend to stretch the resemblance. They might (unconsciously) deduce that the new concept has some properties shared by the familiar concept.
(2) Perceiving a new concept as a process rather than as an object. This is a reflection of the process-object duality (Sfard, 1991). At a lower level, a new concept is perceived as a process which acts on other objects. Only when a higher level of abstraction is reached can the new concept be grasped as an object in itself on which other objects can act.
(3) Reducing the complexity of the new concept, for example, by dealing only with a special case.

The four patterns described above can be mapped to these three forms of reducing the level of abstraction:

- Flattening serves to reduce the complexity of the concept (3) by ignoring some of its dimensions.
- Specializing also reduces the complexity (3).
- Concretizing serves to reduce the complexity of a concept (3). Furthermore, concretizing expresses grasping a concept as a process rather than as an object (2), since concrete instructions induce the process of execution of the program.
- Parallelizing is interpreting a concept as a familiar one (1).

While these patterns were observed for all concepts, their effect might be larger when the gap between the concept and its implementation in a Scratch program is larger. This would explain why the students' performance was lower on the concepts of concurrency, variables, and repeated execution, as explained above.

Abstraction is indeed a central CS skill that is necessary for the CS learning process. According to Piaget's theory of cognitive development (Santrock, 2004), only at about the age of middle-school students do children enter the formal operational stage which enables abstract reasoning. This means that it might be difficult for them to perceive abstract concepts and thus explain their tendency to reduce abstraction.

7.4. The appropriateness of Scratch and similar educational environments

Scratch motivates students by enabling them to create vivid animations and games that involve many sprites. However, projects that involve many

sprites necessarily involve concurrency. Concurrency brings the need for synchronization mechanisms, which in turn necessitates event handling. It seems that the need to cope with advanced and abstract CS concepts is inherent to the environment although it was designed with a young audience in mind. Our findings show that the students can handle some of these concepts, such as communication by messages and event handling, reasonably well, but have difficulties with the more abstract concept of concurrency. If these difficulties are robust, this raises a concern regarding the appropriateness of Scratch and similar environments in which concurrency is unavoidable as tools intended for young children. We believe that these difficulties are not robust and can be overcome. Yet, when considering the use of such environments, one must take into consideration that the teaching and learning processes must cover very advanced concepts, and this, in turn, will affect pedagogical decisions, such as curricular design or the age of the target audience.

7.5. Explicit instruction of CS concepts

We believe that appropriate instruction can improve the learning of CS concepts. Despite its flashy exterior, Scratch is a sophisticated software system and it takes time to learn its technical and pedagogical aspects. The teachers themselves reflected on this issue. In the first year, the teachers (a CS teacher and a mathematics teacher) did not have much time to prepare for the course. The CS teacher said that normally she prepares a course during the summer vacation, but in this case she was asked to teach this course after the school year had begun.

We can demonstrate this for the specific concept of variables. We believe that *explicit* instruction of the concept is needed, but for reasons relating to the teachers' background and time constraints, such instruction was not available to the extent required. The mathematics teacher introduced this concept to her students by saying: "a variable, as you already know from mathematics, can be assigned a value," while the CS teacher merely said "it is something that contains data which we can use." Learning concepts often requires adequate exposure to several instances (Dancik & Kumar, 2003), so we believe that with adequate instruction, the students' learning of this concept and others can improve.

The achievements of students on the concept of variables showed an improvement at the cognitive level of multistructural understanding from the first year (a grade of 8) to the second year (a grade of 69). The teachers in the second year had more time to prepare the course: for two of them it was their second year of teaching the course, and all except one of the new teachers participated in formal training in Scratch during the preceding summer.

7.6. The integrated taxonomy

The design of the research tools and the data analysis used a taxonomy that combined the Bloom and SOLO taxonomies. Our findings indicate that this combined taxonomy captures cognitive characteristics of CS practice. Yet, one important anomaly was observed: in both years, achievements in the high cognitive category of relational applying were much higher than in the lower category of multistructural creating. Tracking code (applying from the Bloom taxonomy) is traditionally considered easier for students than creating code (Fuller et al., 2007; Lister et al., 2006; Robins, Rountree, & Rountree, 2003), but – according to the rationale of our taxonomy – the magnitude of the code (simple or complex), which is expressed in the SOLO abilities it requires (multistructural or relational), is more influential in determining the difficulty of the task.

We conjecture that the categories do not have strict boundaries, but rather allow a continuum of development between cognitive categories. For questions in a boundary area, other factors, such as the type of task or the concepts it involves, may overrule the effect of SOLO complexity. This is an area for future research.

7.7. Threats to validity

The experimental setup was less than optimal in two aspects: While the course was not an elective since every student in these classes had to participate, it was not required by the curriculum, and thus the teachers and students probably felt less committed to this course compared with required courses such as mathematics. In addition, we did not have sufficient time to fully train the teachers before the beginning of the first school year. Nevertheless, we believe that the authentic in-school setting gives our research a high level of ecological validity.

8. Conclusions

We investigated the use of Scratch to teach concepts of CS. We developed new learning materials based upon the constructionist philosophy of Scratch and used them in middle-school classrooms. The research results showed that most students were able to achieve a reasonable level of understanding of CS concepts. Difficulties were encountered in teaching specific topics, such as repeated execution, variables, and concurrency. These difficulties might be rooted in the tendency of students to reduce the level of abstraction, but we believe that the difficulties can be solved by careful teaching, where the relationships between concepts and their implementation in language constructs are taught explicitly and in detail.

Overall, Scratch has proved to be a viable platform for teaching CS, but we do not believe that effective learning will be achieved by itself without

close and effective mentoring. Left to themselves, many students will only use Scratch as a tool for creating media and learn very little. A subsequent publication will describe a study of students who continued from Scratch to the study of standard CS in high schools; the knowledge they learned when studying Scratch turned out to be robust and to facilitate learning CS at a higher level.

We developed a novel taxonomy by integrating the revised Bloom taxonomy and the SOLO taxonomy. This taxonomy enabled us to overcome some of the obstacles in the use of the existing taxonomies in CS education research. We hope that the taxonomy will be used and refined by other researchers.

Acknowledgments

This research was partially supported by the Israel Science Foundation grant 09/1277 and by a Sir Charles Clore Postdoctoral Fellowship. The authors thank Fatima Kaloti-Hallak for her help with this research. The comments of the anonymous referees significantly improved the article.

Notes

1. These data are available at http://stats.scratch.mit.edu/community/blocks.html by hovering over the bars in the histogram.
2. We use the category names from the revised taxonomy, while Lister and Leaney used the original taxonomy.
3. We view a strict hierarchy as desirable in order to evaluate progress in students' learning, but we are willing to relax the hierarchy should that prove fruitful.
4. The Scratch group at MIT has developed their own curriculum guide: http://scratched.media.mit.edu/resources/scratch-curriculum-guide-draft.
5. The textbook can be downloaded under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License from http://stwww.weizmann.ac.il/g-cs/scratch/scratch_en.html.
6. The tests have been translated into English and are available at http://stwww.weizmann.ac.il/g-cs/scratch/tests-cs-concepts-in-scratch.zip.
7. This analysis was described in the conference paper presented at the 6th International Computing Education Research Conference, Aarhus, Denmark, 2010.
8. All quotes from students and teachers were translated from Hebrew by the first author who is a native speaker of Hebrew and checked by the third author who is a native speaker of English.

References

Almstrum, V. (1999). The propositional logic test as a diagnostic tool for misconceptions about logical operations. *Journal of Computer in Mathematics and Science Teaching*, *18*, 205–224.

Anderson, L. W., Krathwohl, D. R., Airasian, P. W., Cruikshank, K. A., Mayer, R. E., Pintrich, P. R., … Wittrock, M. C. (2001). *A taxonomy for learning, teaching and assessing: A revision of Bloom's taxonomy of educational objectives*. New York, NY: Addison-Wesley Longman.

Bell, T., Witten, I. H., & Fellows, M. (2005). *Computer science unplugged*. Retrieved April 3, 2013, from http://csunplugged.com/

Ben-David Kolikant, Y. (2001). Gardeners and cinema tickets: High school students' preconceptions of concurrency. *Computer Science Education*, *11*, 221–245.
Ben-David Kolikant, Y. (2004). Learning concurrency: Evolution of students' understanding of synchronization. *International Journal of Human Computers Studies*, *60*, 259–284.
Biggs, J. B., & Collis, K. F. (1982). *Evaluating the quality of learning: The SOLO taxonomy (Structure of the Observed Learning Outcome)*. New York, NY: Academic Press.
Bloom, B. S., Mesia, B. B., & Krathwohl, D. R. (1964). *Taxonomy of educational objectives (2 volumes: The affective domain; the cognitive domain)*. New York, NY: David McKay.
Carter, L. (2006). Why students with an apparent aptitude for computer science don't choose to major in computer science. *SIGCSE Bulletin*, *38*, 27–31.
Curzon, P., & McOwan, P. W. (2008). Engaging with computer science through magic shows. *SIGCSE Bulletin*, *40*, 179–183.
Dancik, G., & Kumar, A. N. (2003, November 5–8). A tutor for counter-controlled loop concepts and its evaluation. In *Proceedings of the Frontiers in Education Conference*. Boulder, CO. Session T3C.
Dann, W., Cooper, S., & Pausch, R. (2009). *Learning to program with Alice* (2nd ed.). Upper Saddle River, NJ: Pearson.
Fuller, U., Johnson, C. G., Ahoniemi, T., Cukierman, D., Hernán-Losada, I., Jackova, J., … Thompson, E. (2007). Developing a computer science-specific learning taxonomy. *SIGCSE Bulletin*, *39*, 152–170.
Hazzan, O. (2003). How students attempt to reduce abstraction in the learning of mathematics and in the learning of computer science. *Computer Science Education*, *13*, 95–122.
Herman, G. L., Loui, M. C., & Zilles, C. (2010). Creating the digital logic concept inventory. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education (SIGCSE '10)* (pp. 102–106). New York, NY: ACM.
Johnson, C. J., & Fuller, U. (2006, November 9–12). Is Bloom's taxonomy appropriate for computer science? In *Proceedings of the 6th Baltic Sea Conference on Computing Education Research: Koli Calling*, Uppsala, Sweden (pp. 120–123). New York, NY: ACM.
Johnson, R. B., & Onwuegbuzie, A. J. (2004). Mixed methods research: A research paradigm whose time has come. *Educational Researcher*, *33*, 14–26.
Kelleher, C., & Pausch, R. (2005). Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys*, *37*, 83–137.
Kelleher, C., Pausch, R., & Kiesler, S. (2007, April 28–May 3). Storytelling Alice motivates middle school girls to learn computer programming. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, San Jose, CA (pp. 1455–1464). New York, NY: ACM.
Krathwohl, D. R. (2002). A revision of Bloom's taxonomy: An overview. *Theory into Practice*, *41*, 212–264. Retrieved April 3, 2013, from http://www.unco.edu/cetl/sir/stating_outcome/documents/Krathwohl.pdf
Kuittinen, M., & Sajaniemi, J. (2004). Teaching roles of variables in elementary programming courses. *SIGCSE Bulletin*, *36*, 57–61.
Lambert, L., & Guiffre, H. (2009). Computer science outreach in an elementary school. *Journal of Computing in Small Colleges*, *24*, 118–124.
Lister, R., & Leaney, J. (2003). Introductory programming, criterion-referencing and Bloom. *SIGCSE Bulletin*, *35*, 143–147.
Lister, R., Simon, B., Thompson, E., Whalley, J. L., & Prasad, C. (2006). Not seeing the forest for the trees: Novice programmers and the SOLO taxonomy. *SIGCSE Bulletin*, *38*, 118–122.
Malan, D. J., & Leitner, H. H. (2007). Scratch for budding computer scientists. *SIGCSE Bulletin*, *39*, 223–227.
Maloney, J., Peppler, K., Kafai, Y., Resnick, M., & Rusk, N. (2008). Programming by choice: Urban youth learning programming with Scratch. *SIGCSE Bulletin*, *40*, 367–371.

Meerbaum-Salant, O., Armoni, M., & Ben-Ari, M. (2011). Habits of programming in Scratch. In *Proceedings of the Sixteenth SIGCSE Conference on Innovation and Technology in Computer Science Education, Darmstadt, Germany*, pp. 168–172.

Moskal, B., Lurie, D., & Cooper, S. (2004, March 3–7). Evaluating the effectiveness of a new instructional approach. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*, Norfolk, VA (pp. 75–79). New York, NY: ACM Press.

Papert, S. (n.d.). *Hard fun*. Retrieved March 28, 2013, from http://www.papert.org/articles/HardFun.html

Papert, S., & Harel, I. (1991). *Constructionism*. Norwood, NJ: Ablex.

Paul, J. L., Kouril, M., & Berman, K. A. (2006). A template library to facilitate teaching message passing parallel computing. In *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '06)* (pp. 464–468). New York, NY: ACM.

Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., … Kafai, Y. (2009). Scratch: Programming for all. *Communications of the ACM*, *52*, 60–67.

Robins, A., Rountree, J., & Rountree, N. (2003). Learning and teaching programming: A review and discussion. *Computer Science Education*, *13*, 137–172.

Samurçay, 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 Associates.

Santrock, J. W. (2004). *Life-span development* (9th ed.). Boston, MA: McGraw-Hill College.

Schwill, A. (2004). Philosophical aspects of fundamental ideas: Concepts and ideas. In J. Magenheim, & S. Schubert (Eds.), *Informatics and student assessment – concepts of empirical research and the standardisation of measurement in the area of didactics of informatics, lecture notes in informatics* (Vol. 1, pp. 145–157). Bonn: Köllen Druck + Verlag.

Sfard, A. (1991). On the dual nature of mathematical conceptions: Reflections on processes and objects as different sides of the same coin. *Educational Studies in Mathematics*, *22*, 1–36.

Taub, R., Armoni, M., & Ben-Ari, M. (2012). CS unplugged and middle-school students' views, attitudes, and intentions regarding CS. *ACM Transactions on Computing Education*, *12*, 1–29.

Thompson, E., Luxton-Reilly, A., Whalley, J. L., Hu, M., & Robbins, P. (2008, January 20–23). Bloom's taxonomy for CS assessment. In *Proceedings of the Tenth Conference Australasian Computing Education*, Wollongong, NSW, Australia, pp. 155–161.

Whalley, J., Lister, R., Thompson, E., Clear, T., Robbins, P., & Prasad, C. (2006, January 16–19). An Australasian study of reading and comprehension skills in novice programmers, using the Bloom and SOLO taxonomies. In *Proceedings of the Eighth Australasian Conference Computing Education*, Hobart, Australia, pp. 243–252.

Wilson, A., & Moffat, D. C. (2010, September 19–22). Evaluating Scratch to introduce younger schoolchildren to programming. In *Proceedings of the 22nd Annual Workshop of the Psychology of Programming Interest Group*, Madrid, Spain, pp. 63–75.

Wolz, U., Leitner, H. H., Malan, D. J., & Maloney, J. (2009). Starting with Scratch in CS1. *SIGCSE Bulletin*, *41*, 2–3.

Wolz, U., Maloney, J., & Pulimood, S. M. (2008). "Scratch" your way to introductory CS. *SIGCSE Bulletin*, *40*, 298–299.