

Exploring the intersection of algebraic and computational thinking

Kajsa Bråting & Cecilia Kilhamn

To cite this article: Kajsa Bråting & Cecilia Kilhamn (2021) Exploring the intersection of algebraic and computational thinking, *Mathematical Thinking and Learning*, 23:2, 170-185, DOI: [10.1080/10986065.2020.1779012](https://doi.org/10.1080/10986065.2020.1779012)

To link to this article: <https://doi.org/10.1080/10986065.2020.1779012>



© 2020 The Author(s). Published with license by Taylor & Francis Group, LLC.



Published online: 21 Jun 2020.



Submit your article to this journal [↗](#)



Article views: 6464



View related articles [↗](#)



View Crossmark data [↗](#)



Citing articles: 3 View citing articles [↗](#)



Exploring the intersection of algebraic and computational thinking

Kajsa Bråting ^a and Cecilia Kilhamn ^b

^aDepartment of Education, Uppsala University, Uppsala, Sweden; ^bDepartment of Pedagogical, Curricular and Professional Studies, University of Gothenburg, Gothenburg, Sweden

ABSTRACT

This article investigates how the recent implementation of programming in school mathematics interacts with algebraic thinking and learning. Based on Duval's theory of semiotic representations, we analyze in what ways syntax and semantics of programming languages are aligned with or divert from corresponding algebraic symbolism. Three examples of programming activities suggested for school mathematics are discussed in detail. We argue that although the semiotic representations of programming languages are similar to algebraic notation the meanings of several concepts in these two domains differ. In a learning perspective these differences must be taken into account, especially considering that students have to convert between registers with both overlapping and specific meanings.

ARTICLE HISTORY

Received 12 December 2019
Revised 2 June 2020
Accepted 3 June 2020

KEYWORDS

Algebraic thinking;
computational thinking;
semiotic representation;
programming; mathematics
education

Introduction

While algebra has a long history, both as a research field in mathematics and as content taught in schools, computer science is a recently developed field of knowledge. Along with a fast-changing and increasing digital society, programming and computational thinking have emerged as necessary skills not only for computer scientists and engineers but also for all citizens. This article sets out to explore some similarities and differences in the ways of thinking required in these two fields. To do this, we specifically investigate the different systems of representations that are used and how these relate to each other.

The term computational thinking was first mentioned by the *Logo* programming developer Seymour Papert, who believed that programming enables pupils to develop mathematical understanding through the process of testing and debugging their own ideas in a code (Papert, 1980; 1996). Papert's ideas did not have any major impact at the time, possibly because digital technology had not become a natural part of people's daily lives as it has today through the internet and the abundance of digital devices (Kotsopoulos et al., 2017). However, about thirty years later, Jeanette Wing (2006) returned to the term, arguing that computational thinking should be taught in schools alongside reading, writing and arithmetic. Although computational thinking may seem more inclusive than the term programming suggests, we focus our discussion in this article on aspects of computational thinking that can be developed through programming.

Today, computational thinking has made its way into the educational system and school curricula in many countries, e.g., Australia (Falkner et al., 2014), England (Brown et al., 2014), Finland (Mannila et al., 2014), Sweden (Kilhamn & Bråting, 2019) and the United States (Fisher, 2016). This integration has been done in various ways. For instance, in England, programming was made part of a new subject, "Computing", while Finland and Sweden adopted a blend of cross-curriculum and single subject integration with the strongest link to mathematics (Bocconi et al., 2018).

20th century efforts in promoting programming in schools were all focused on using programming as a tool to explore and express mathematical ideas. As various programming languages and micro-world environments inspired by Logo emerged, educational research tried to show in what ways students improved their mathematical thinking skills or understanding of mathematical concepts and relationships through programming (Hoyles & Noss, 2003; Noss & Hoyles, 1996). Another type of digital technologies that appeared at around the same time were expressive tools such as dynamic geometry systems and computer algebra systems. In 2003, Hoyles and Noss noted that programming microworlds were increasingly being incorporated into expressive tools. Drawing on both these types of digital technologies, programmers at Massachusetts Institute of Technology developed a visual programming environment called Scratch, which was publicly launched in 2007 and quickly grew in popularity.¹ The intended goal was “to make it easy for everyone, of all ages, backgrounds, and interests, to program their own interactive stories, games, animations and simulations, and share their creations with one another” (Resnick et al., 2009, p. 60). This goal is far removed from the early ideas of Logo where the connection to mathematics was promoted. The launch of Scratch coincides with the worldwide introduction of programming in school curricula, very often as part of the mathematics curriculum.

Programming and algebra

Unlike other countries, Sweden implemented programming in the mathematics curriculum within the core content of algebra through all grade levels, which makes the Swedish case unique in an international perspective (Bråting et al., 2020). Until now, research on computational thinking and algebraic thinking has run on separate tracks, but the Swedish case offers a great opportunity to investigate the intersection of these two research domains.

The aim of the present paper is to explore how semiotic representations related to computational thinking compare and interact with algebraic thinking and, ultimately, how this may affect students’ learning of algebra. Within the research field of algebraic thinking, the connection between the learning of algebra and computer programming is rarely discussed. For instance, in a state-of-the-art publication on the teaching and learning of early algebra from ICME 13 (Kieran, 2018) the word “programming” is never mentioned and “computer” appears only in one paragraph, where John Mason writes:

[L]anguages that are both expressive and readily manipulable are emerging from algebra into the domain of computer technology. LOGO, Boxer, TuneTalks, computer algebra systems, and more recent offshoots, such as TouchCounts, provide people of different ages and maturity with expressive and manipulable languages. All of these invoke algebraic thinking and algebraic awarenesses (Mason, 2018, p. 335).

While Mason may be right in the assumption that different programming languages may invoke algebraic thinking and algebraic awareness, we want to draw attention to the way this is done and what the consequences could be in terms of learning algebra. Programming languages can be seen as new systems of representation used to represent mathematical objects and structures. However, while the primary function of programming languages is to facilitate data processing and calculations, the objects represented are not always mathematical and the kind of thinking involved may not always be algebraic.

In this paper, our discussion is centered around an analysis of three different programming activities suggested for school use by Swedish government-provided on-line materials for professional development.² To deepen our analysis, we utilize Duval’s (2006) theoretical framework regarding semiotic systems of representation. In particular, we highlight syntactic and semiotic aspects of algebraic concepts that appear in both algebra and programming, such as equality, variable, algorithm and function. We reveal potential conflicting interpretations when these concepts appear in the different systems of representation of these domains. Especially, we investigate how programming-related representations interact with traditional algebraic systems of representation.

We pose the following two research questions:

- (1) *What differences and similarities can be seen in how variables, equality, functions and algorithms are represented in computer programs and algebraic notation?*
- (2) *How can the integration of programming into school mathematics potentially interact with, afford or constrain students' development of algebraic thinking?*

Algebraic thinking and computational thinking

In this section we give a brief description of algebraic thinking (AT) and computational thinking (CT) and discuss some issues of the intersection between the two fields that could need to be further researched.

Algebraic thinking is a broad term encompassing types of reasoning and ways of representing used when engaging in algebraic activities. Over the years, algebra has made its way from higher education down through the school system (Kieran et al., 2016) and many researchers have tried to define algebra in a school context (e.g. Bednarz et al., 1996; Blanton et al., 2015; Kilhamn et al., 2019). Although there are differences, most frameworks identify similar activities and skills as important to include in school algebra. For example, Kaput (2008) describes early algebra in terms of the three content strands: i) the study of structures; ii) the study of functions; and iii) the application of a cluster of modeling languages both inside and outside of mathematics. As a modeling language algebra is permeated with symbols, and Arcavi (1994) introduced the term “symbol sense” to describe the kind of knowledge he considered to be at the heart of what it means to be competent in algebra.

One of the most condensed definitions of algebraic thinking is suggested by Radford (2018, p. 8):

[A]lgebraic thinking resorts to: (a) indeterminate quantities and (b) idiosyncratic or specific culturally and historically evolved modes of representing/symbolizing these indeterminate quantities and their operations, and (c) deals with indeterminate quantities in an analytical manner.

Although Radford does not see an alphanumerical symbol system as the only way of expressing generalizations, he does emphasize the importance of symbolizing and representations of unknowns. When considering this definition in the light of programming, the question that arises is if a programming language, which undoubtedly is a culturally developed system of representations, can be used to foster thinking about indeterminate quantities in an analytical manner. Indeterminate quantities include all forms of unknown, arbitrary or varying numbers. In algebra these are sometimes all referred to as variables, sometimes described in more detail, for example, by Usiskin (1988) as pattern generalizers; unknowns and constants; arguments and parameters; or as arbitrary marks that can be manipulated. In an extensive overview of research about algebra skills and misconceptions, Bush and Karp (2013) found that many students struggle with algebraic expressions and the meaning of variables. Some researchers therefore advocate a separation between the different meanings, for example, Ely and Adams (2012) suggest clear distinctions between unknowns, placeholders and variables. According to Usiskin (1988), variables in computer science cover all the uses of variables in algebra.

Computational thinking (CT) is a fairly new concept in educational research, first introduced by Papert in 1996. Wing (2006) re-introduced the term as a fundamental skill for everyone, an essentially human way of thinking that enables us to make use of computers to solve problems (see also Grover & Pea, 2013). Brennan and Resnick (2012) developed a CT framework comprising three dimensions: concepts, practices and perspectives. Computational concepts are, for example, sequences, loops, conditionals and data. Variables are mentioned as a type of data and a way of storing data in the computer's memory (ibid). Computational practices refer to programming practices such as testing, debugging, reusing and remixing and abstracting. One of the core features of CT is the use of algorithms and the development of algorithmic thinking (Wu & Richards, 2011). Sometimes the latter appears as a synonym to computational thinking (e.g., Stephens, 2018). Futschek (2006) defines an

algorithm in computer science as “a method to solve a problem that consists of exactly defined instructions” (p 160), with algorithmic thinking as a pool of abilities that are connected to constructing and understanding algorithms. Denning (2017) argues that the meaning of the terms algorithm and computational thinking is sometimes misused in educational contexts. He points out that an algorithm in computer science consists of a series of steps that controls a machine or a computational model and that CT includes designing the model, not only the steps to control it. He highlights Aho’s (2012) definition of computational thinking as “the thought processes involved in formulating problems so their solutions can be represented as computational steps and algorithms” (p. 832).

Although CT is generally considered to encompass more than programming, teaching and learning programming requires the use of CT (Hickmott et al., 2018). Programming does not necessarily involve writing code and is thus a more inclusive term than coding (Bocconi et al., 2018). However, many definitions of computational thinking highlight symbolization and the understanding of symbol systems and representations (e.g., Grover & Pea, 2013; Kilhamn & Bråting, 2019). Stephens (2018) describes programming as developing a logic-focused mind-set, and coding as a formalized means of recording and executing algorithms. For the purpose of this paper, and in line with Mannila et al. (2014), we consider programming as an activity in which students develop CT.

Much research about programming in educational settings concerns the learning of programming per se, for instance, students’ misconceptions and other difficulties in introductory programming courses (e.g., Qian & Lehman, 2017). Contemporary research focusing on learning mathematical ideas through programming is on the other hand rare. One exception is research related to the ScratchMath project in the United Kingdom, where Benton et al. (2017) found that programming activities could broaden students’ learning of mathematical ideas such as place value, proportional relationships, coordinate systems, symmetry and negative numbers. However, when these activities were tried out in a professional development program focusing on the integration of CT within mathematics education in Australian primary schools, teachers found that students did not engage in the mathematical concepts underpinning the activities (Holmes et al., 2018). These results mirror how important ideas in Logo disappeared from view as the program filtered down through the educational system and became reduced to turtle graphics “with little emphasis on any aspect of mathematics or even geometry, let alone of programming as a means of mathematical expression” (Noss & Hoyles, 1996, p. 162).

Malara and Navarra (2018) describe algebraic thinking (AT) as shift of attention from the result to the process in problem-solving situations. Computational thinking (CT) is also centered around problem-solving processes, using debugging and tinkering as practices that explore the structure of an algorithm. From very different angles, both domains address the structure of a process more than its result, at least on a theoretical level.

Even if computer languages originated from algebra, computer environments bring along new systems of representation with syntactic rules that are different from those of natural language or of algebraic notation. Malara and Navarra (2018) argue that to learn a new representational language, it is necessary to focus on both syntax and semantics. In a study of university level physics education, Sherin (2001) investigated the implications of replacing algebraic notation with a programming language as the principal representational system for physics instruction. Students were exposed to either algebra-based or programming-based physics instruction. Sherin found that the use of the two symbol systems provided the students with different conceptualizations of physics, as either a physics of balance and equilibrium when based on algebra, or as a physics of processes and causation when based on programming. This is an example of the difference between algebra as more static and programming as dynamic and process-oriented.

In school mathematics, the idea and use of algorithms have changed greatly since the introduction of digital tools. Before 1980, traditional algorithms were seen as a cornerstone of arithmetic, but following the invention of pocket calculators a debate flourished on the necessity of these algorithms (Kamii & Dominick, 1997). Traditional algorithms were replaced by an increased emphasis on number sense and conceptual understanding. In Sweden, the term algorithm was removed from the

description of arithmetic in the national curriculum in 2011, but re-inserted in 2017 as an aspect of the core content of algebra in connection to programming (Kilhamn & Bråting, 2019; Swedish National Agency for Education, 2017). This change implies a shift of emphasis from a procedural use of algorithms, to a structural understanding of algorithms in terms of algorithmic thinking as defined by Futschek (2006).

Theoretical framework

We consider here the notion of semiotic representation to mean signs of various modes, along with the syntactic rules applied to these signs, which make them useful as descriptions of phenomena, processes and relations (Duval, 2006). Natural language, symbolic algebra, cartesian graphs and various computer languages are examples of different semiotic systems of representation. In this paper we are concerned with mathematical objects, defined by Duval (p 129) as “*the invariant of a set of phenomena or the invariant of some multiplicity of possible representations*”. Semiotic representations are not mathematical objects in themselves. Take, for example, an equation of a line. The equation itself is a phenomenological object, but it represents a mathematical idea of a straight line with certain properties, which could also be represented by, for example, a graph. The equation and the graph are signs that stand for a mathematical object that is invariant in the two representations. Although the two representations stand for the same mathematical object, the different systems of representation will give access to different properties of the mathematical object. Also, some processes are easier in one system of representation than in another, or perhaps only possible in one and not the other. At the heart of mathematics is the ability to substitute some signs for other signs. However, according to Duval (2006), a substitution of signs is rarely a simple translation or encoding, since new ways of representing reveal new properties and enable different processes. He therefore uses the term transformation when some signs are substituted for others, within or between systems of representations. Although the object stays the same, its appearance is transformed. In line with Duval, we will use the term *register* to mean a system of representations that permit transformations.

Mathematical objects, claims Duval (2006), are only accessible through registers and are generally understood through processes of transformation. When learning mathematics, at least two different registers are evoked simultaneously since natural language is used to explain more specifically mathematical representations. This means that in a learning process, a student is faced with the challenge of simultaneously applying a new register, making transformations between registers, and learning about the mathematical objects being dealt with. Consequently, transformation between registers (e.g., between natural language and algebraic notation) and between representations within the same register (e.g., equivalent equations) is an integrated part of learning mathematics.

Duval (2006) labels transformation between registers as *conversions*, and within the same register as *treatments*. Some conversions are congruent, and similar to encoding. An example of a congruent conversion is when the sentence *Five plus four equals nine* in natural language is translated into symbolic notation as $5 + 4 = 9$. More often, however, conversions are non-congruent. If the sentence is changed only slightly, to *Adding five and four gives the sum of nine*, the word *adding* comes first, rather than between the numbers. Non-congruent conversions are, according to Duval, the cause of most problems for a mathematics learner.

Registers can have different properties. Some, such as algebraic notation, are exclusively used for mathematical processes like computations and proofs. Within such registers, processes can take the form of algorithms. Other registers, such as natural language, images, iconic drawings and geometrical constructions, are multi-functional in the sense that they “*can fulfill a large range of cognitive functions: communication, information processing, awareness, imagination etc*” (Duval, 2006, p. 109). In such registers, he claims that processes can never be converted into algorithms. In a thorough analysis of programming languages, Gazoni (2018) finds that the most important semiotic difference between programming languages and natural languages lies in the fact that the former does not bear any vagueness. The vagueness Gazoni describes is what Duval

calls multi-functionality. The point of a programming language is to remove vagueness and build all communication on algorithmic processes.

A crucial idea in Duval's theory about semiotic systems of representations is that a sign can only function within its own register, where the meaning of the sign is conveyed through its relations to other signs in the system. Take the letter x , it is a sign with different meanings in a linguistic register from an algebraic register, since different rules apply for it when it is used to spell words and when it is used to express algebraic relationships.


In addition to natural language, we will look at *algebraic notation* (alphanumerical symbol systems including arithmetic symbols), as well as the different registers introduced in each specific programming activity.

Examples

In this section, we will analyze three examples of programming activities from different grade levels suggested in the afore-mentioned government-provided teaching material in Sweden. The material reflects the revised Swedish mathematics curriculum which is structured according to the three grade levels 1–3, 4–6 and 7–9. In grades 1–3 the material, as well as the curriculum, focus on the use of symbols to construct and follow stepwise instructions. In grades 4–6 algorithms are created in visual programming languages and in grades 7–9 text-based programming languages are introduced. The material includes activities that can be used in the mathematics classroom and is intended to inspire teachers and support teachers introducing programming in mathematics. All three examples considered in this paper can be seen as typical for each of the grade levels based on the Swedish revised curriculum. In our analysis, we will highlight points of intersection between programming and algebra and consider these in connection to Duval's (2006) theory. In particular, we will analyze syntactic rules and the meaning of the concepts algorithm, equality, variable, and function within registers related to programming and algebra.

Example 1: Lightbot, a programming game

This example is directed to grades 1–3 when students learn the basics of programming by means of stepwise instructions. The activity uses the commercially accessible application Lightbot.³ According to the Lightbot web page students learn: Sequencing, overloading, procedures, recursive loops, and conditionals. Using iconic symbols for step, jump, turn right, turn left, switch on light (Figure 1), the aim of the game is to guide a robot in different landscapes of square tiles so that it will switch on the lights on specific blue tiles. A *sequence of commands* can be organized in procedures, named MAIN, PROC1 and PROC2. MAIN is always executed and to call PROC1 or PROC2 the *commands* P1 and P2 can be used. That is, when given the command P1 the robot will do all the commands embedded in PROC1.

Let us take a closer look at the particular task given in Figure 1, where the challenge is to program the robot to switch on the light on all the blue tiles using only one command in the MAIN procedure. To do that, the player must construct two procedures, PROC1 and PROC2. To solve the task the player needs to detect a pattern, in this case the four identical edges in Figure 1. Based on the detected pattern, the player then constructs the procedure PROC2 using the stepwise instructions “switch on light, step forward, switch on light, step forward, switch on light, jump, switch on light, turn right” so that PROC2 performs the subtask of lighting one side of the square. The player can now solve the whole task by repeating PROC2 four times. This is performed in PROC1 with the instructions “jump, P2, jump, P2, jump, P2, jump, P2”. Note that PROC2 is embedded in PROC1 as the command P2. In terms of a semiotic system, PROC2, P2 and  can all be seen as different representations of the same procedure within the game. To solve the task the player must be able to substitute between these representations. In Duval's terms, a treatment is made when transforming one representation into another within the same system. In this case, the treatment is a non-congruent transformation of a longer string of symbols into a shorter, more dense representation.

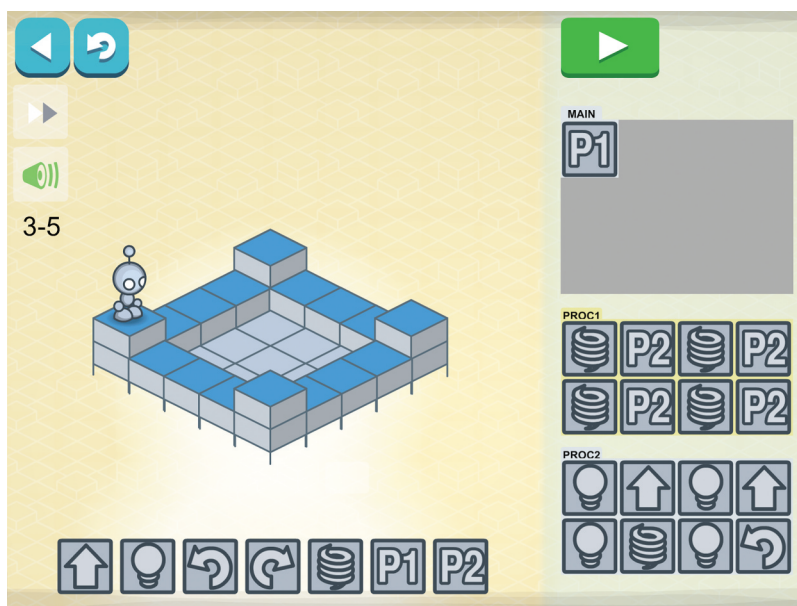


Figure 1. The Lightbot landscape of square tiles and the sequence of commands in Example 1.

In Lightbot, procedures are described as “sets of instructions that are needed to solve the task” (lightbot.com). This description is aligned with how algorithms are described in computer science (Aho, 2012; Denning, 2017; Futschek, 2006). It resembles what is treated as an algorithm in mathematics, but looking closely there are also significant differences. Brousseau’s definition of an algorithm is commonly referred to in mathematics education: “[...] a finite sequence of executable instructions which allows one to find a definite result for a given class of problems” (Brousseau, 1997, p. 130). The most striking similarity is that both descriptions can be viewed as “a set of finite instructions”. However, Lightbot procedures cannot be applied to “find a definite result for a given class of problems”; they can only accomplish one single task. Our point is that in Lightbot students solve specific prepared tasks and not mathematical problems. In Lightbot tasks, variables are not included. The problem in Figure 1 is a typical example of a task that can be solved using stepwise instructions, but the procedure cannot be reused to solve another task or a class of problems. In order to create an algorithm to solve a general mathematical problem, we need to create and use variables as we will see in Examples 2 and 3, where almost every line in the code includes one or more variables.

In programming, the meaning of procedures and functions is sometimes similar. For instance, in the “Hour of code” tutorial a function is described as “[...] a specific set of instructions to accomplish a certain task, kind of like a recipe”,⁴ which is almost the same as the description of a procedure in Lightbot as cited above. These two descriptions differ significantly from the meaning of a function in mathematics, even though we should bear in mind that Hour of code and Lightbot are adapted to children. The main difference between a mathematical function and a function in programming is that the latter can have *side effects*, meaning that the function modifies some state outside its local environment or has an observable interaction with the outside world besides returning a value (Spuler & Sajeev, 1994). In Lightbot, the side effects consist of a robot switching on the lights on specific blue tiles in a certain landscape. A mathematical function cannot have side effects, instead, what characterizes a mathematical function is that it always has input and output values and is an injective relation. In the Lightbot example there are no input and output values at all, only commands and side effects. The term function is used in both computer-related natural language and algebra-related mathematical language but does not represent entirely the same object in the two registers. Both similarities and differences between a mathematical function and a function in programming are

important to take into consideration in a learning situation. Especially in the case when the intention is to use programming in order to learn mathematics. We return to the function concept in Example 2 below.

To summarize, two important features of the intersection between AT and CT appeared in this example. One was the similar process of condensing symbolic representations, the other was two examples of semiotic representations (algorithm and function) that referred to slightly different objects in the two domains. Although it is easy to confuse a term with a concept when they are so alike, there are conceptual differences between algorithms and functions in mathematics and in computer science.

Example 2: Investigating multiplication in a visual programming language

Our second example is an activity adapted for students in grades 4–6. The aim is to develop a multiplication machine. The teacher is told to first discuss with the students what instructions would be necessary to make a machine calculate the product of two input values. Since multiplication is commonly introduced as repeated addition, we assume that suggestions in line with that will appear. Hence, the activity could be used to explore a structural understanding of multiplication as repeated addition. Thereafter, the students are to create their own algorithms for calculating a multiplication in the visual programming language Scratch,⁵ pursuing and exploring their original ideas. Since the material does not provide examples of code we have created our own example. Our machine multiplies two numbers by repeated addition and is applicable to non-negative integers (Figure 2ab).

Scratch is a block-based visual programming language and an online environment adapted primarily for children. The Scratch code utilizes variables that enable users to solve classes of problems and not only single tasks, which was the case in the Lightbot example above. The users are supposed to introduce their own variables as in Figure 2a below. In our activity the students need four variables; the two input variables *var1* and *var2*, the loop variable *count*, and finally the output variable *sum*. The program is simple, as Figure 2b shows, it starts to request two numbers, *var1* and *var2*, by utilizing the

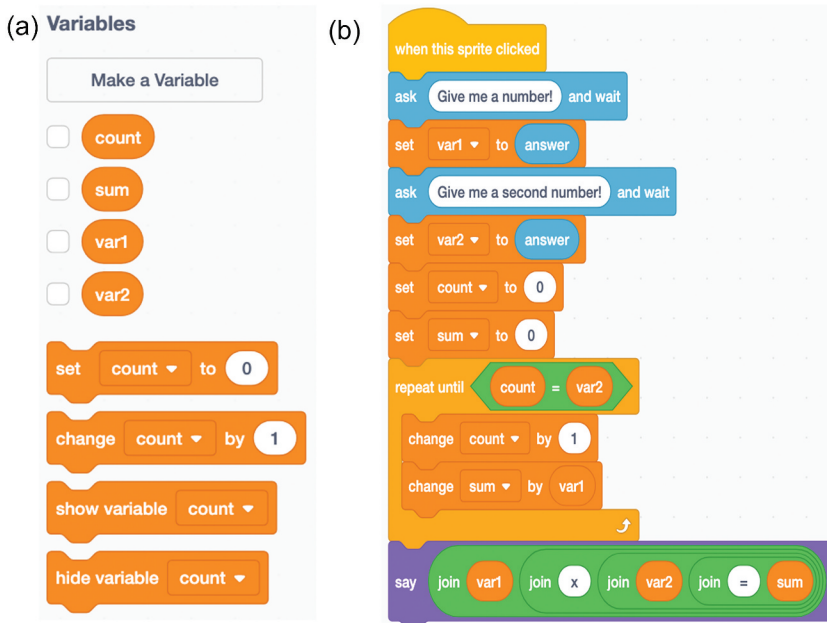


Figure 2. A. The variables in Example 2, B. The scratch code of repeated multiplication.

so called sensing block “ask and wait”. Thereafter the program performs repeated addition of *var1* in the loop block “repeat until” the same number of times as recorded in the variable *var2*. We now take a closer look at the different characteristics of the variables in our multiplication machine. Thereafter we will compare the concept of variable in Scratch with variables in algebra.

The input variables *var1* and *var2* are given values initially in the program by means of the *assignment* block “set *var1* to answer” where “answer” is the value entered by the user of the program. Syntactically, this differs from text-based programming languages that use the equal sign for assignments which we will return to in Example 3 below. Furthermore, the values of *var1* and *var2* are never reassigned, they keep their values throughout the whole program. This is different from the variables *count* and *sum* that change their values during the program. The loop variable *count* is initiated to 0 (Figure 2b) and thereafter increases its value by one in order to keep track of the repeated additions through the block “change *count* by 1”. Correspondingly, the variable *sum* is initiated to 0 and accumulates the repeated additions through the block “change *sum* by *var1*” (Figures 2ab). The final value stored in the variable *sum* is the result of the repeated addition, that is, *sum* is an output variable.

Let us consider how we would represent repeated addition using an algebraic system of representation. There are of course several ways to do that but here we present a way that corresponds closely with the procedures in our multiplication machine. We choose the multiplication $17 \cdot 5$, which by repeated addition can be represented as:

$$17 \cdot 1 = 17 \cdot 0 + 17 = 0 + 17 = 17$$

$$17 \cdot 2 = 17 \cdot 1 + 17 = 17 + 17 = 34$$

$$17 \cdot 3 = 17 \cdot 2 + 17 = 34 + 17 = 51$$

$$17 \cdot 4 = 17 \cdot 3 + 17 = 51 + 17 = 68$$

$$17 \cdot 5 = 17 \cdot 4 + 17 = 68 + 17 = 85$$

Here, the variable *count* corresponds to the number of lines calculated and the variable *sum* to the partial sum of a line, for instance, “34” on line 2. Note that inserting the variable *sum* in one separate line would make it impossible to use *sum* in any other line. For instance, using *sum* in lines 2 and 3 would imply that $34 = 51$, which is a contradiction. Instead, both *sum* and *count* can here only be considered as variables that keep track of the *process* performed within the calculation. In mathematics, the variable *count* is often represented by tallying when keeping track of repeated additions. In programming, we use a piece of the computer’s memory to keep track of how far we have reached in the calculation process. The role of a variable in programming is to keep track of the same piece of memory during the program. This can be contrasted with how we keep track of a series of expressions in mathematics by means of a range of indexed variables, for instance, when representing each successive term in the series $\sum a_n = a_1 + a_2 + \dots + a_k$. That is, in programming we can use a single variable that changes ⁱⁿ the execution of the program, while in mathematics we need a series of indexed variables to accomplish the same thing.

An important aspect of a programming language is that it allows users to solve generic problems in a manner that can be reused. We have already seen that variables are an important part of this process because they enable the solution to be parametrized and capture an entire class of problems. Another important aspect for allowing reuse is to give the solution a name and make it clear how and under what circumstances it can be reused. In contrast to the algorithm in Example 1, the algorithm here solves a class of problems in the same way as a mathematical algorithm (cf. Brousseau, 1997), thanks to the use of the two variables that serve as input values. The algorithm can be used on any non-negative integer value for *var1* and *var2*, but at the same time it is somewhat limited by including a dialog with the user (the blocks “ask” and “say” in Figure 2b). For instance, it cannot be reused when writing

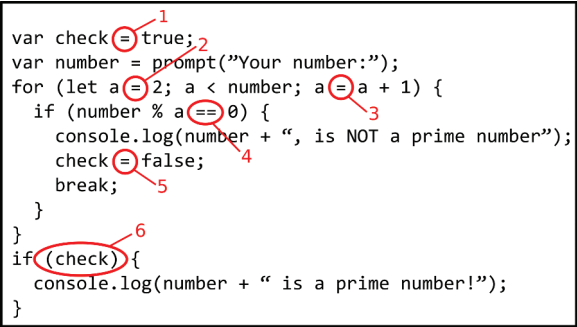
a larger program that solves the task of exponentiation by repeated multiplication because in that scenario you would not need user interaction in each multiplication step. To prepare for such reusability the example of performing multiplication by repeated addition would need to be divided into two parts; one part responsible for the user interaction and another for performing the multiplication. The latter part obviously corresponds to a mathematical function.

To summarize our findings in Example 2, we note that in the register introduced in Scratch it is possible to work with algorithms and functions that behave as their counterparts in mathematics. Since our problem is mathematical it is perhaps not surprising that this procedure behaves like a mathematical function. However, the program also includes a variable of a different kind, which changes its value as the program is executed. This type of variable is not easily converted into algebraic notation since the way of representing the idea is quite different in the two registers. Structures inherent in the idea of multiplication as repeated addition are visible in the Scratch representation, but while it is the structure of the process (i.e. the idea of a loop) that is in focus in programming, the algebraic representation reveals a series of number patterns.

Example 3: Investigating prime numbers in a text-based programming language

In the third activity, adapted to grades 7–9, the task is to write a short program inspired by the algorithm of “Sieve of Eratosthenes”, that is, an algorithm for finding prime numbers. The material supplies an example of an algorithm written in the programming language JavaScript (Figure 3). By using this algorithm, one can find out if a given whole number is a prime number or not. The students need to be familiar with handling variables, functions and the control flow instructions for and if. We have highlighted six places in the code that illustrate some crucial passages regarding syntactic rules and semantics. These places are circled and marked from 1–6 in Figure 3 and will be analyzed next.

Let us begin to consider the equal sign. At the places 1, 2, 3 and 5 in Figure 3 the equal signs all represent *assignments* of different values to variables. For instance, at place 2 the program declares the variable *a* and assigns it the value 2, that is, gives the variable the value 2. This usage of the equal sign differs from how it is normally used in algebra, where the equal sign is a symbol for an equivalence *relation*, and different from arithmetic, where young pupils tend to interpret the equal sign as an *operator* symbol ($5 + 3$ make 8) rather than a relation (Kieran, 1981). Hence, the sign “=” is included in both JavaScript and algebraic notation, although representing different ideas (see also Altadmri & Brown, 2015; Sirkia & Sorva, 2012). Therefore, it is important to keep in mind that a sign can only function and have meaning within its own register (Duval, 2006). In this example, the equal sign should be treated as a symbol for an assignment in JavaScript and as a symbol for an equivalence relation in algebra. This problem does not appear in Example 2 since assignments in Scratch are represented by the instruction block “set to” and not by the equal sign.



```

var check = true;
var number = prompt("Your number:");
for (let a = 2; a < number; a = a + 1) {
  if (number % a == 0) {
    console.log(number + ", is NOT a prime number");
    check = false;
    break;
  }
}
if (check) {
  console.log(number + " is a prime number!");
}

```

The figure shows a JavaScript code snippet for determining if a number is prime. Six specific parts of the code are circled in red and labeled with numbers 1 through 6, with red arrows pointing to them. The annotations are as follows: 1 points to the first '=' in 'var check = true;'; 2 points to the second '=' in 'var number = prompt(...)'; 3 points to the '=' in 'a = a + 1'; 4 points to the '=' in 'number % a == 0'; 5 points to the '=' in 'check = false;'; and 6 points to the parentheses in the 'if (check)' statement.

Figure 3. An algorithm for deciding if a given integer is a prime number.

In algebra, it would be meaningless to write $a = a + 1$, since it is not true for any value of a while in programming it makes sense since the equal sign should be interpreted as a symbolic expression for the assignment “add 1 to the value a ” (see place 3). This is often used when a program needs to loop through a range of consecutive integers, as in [Figure 3](#) when the program loops from 2 to the input value stored in the variable “number”. In this example, the expression $a = a + 1$ corresponds to the instruction block “change *count* by 1” in Example 2. That is, the same idea is represented differently in the two programming languages Scratch and JavaScript, where a conflict with algebraic notation only appears in the latter case. In JavaScript the so called increment operators $a++$ and $a+ =$ also represent the idea “add 1 to the value a ”. Thus, in JavaScript there are several different representations of the same idea, in which the signs $=$ and $+$ appear in slightly different meanings compared to algebraic notation. Therefore, conversions between natural language and symbolic registers will be different if the conversion is to JavaScript or to algebraic notation. In addition, a conversion between JavaScript and algebra entails yet another a non-congruent transformation.

Let us now consider place 4 where we have two consecutive equal signs ($==$). This is the only place in this code where the equal sign is not used as an assignment. Instead, $==$ is used as a relational operator that tests if two entities are equal. In our example the program tests if the result of the modulo operation “number % a ” is equal to 0, that is, if the remainder is equal to 0 after dividing the input value “number” with the loop variable a . Here, the meaning of the double equal sign ($==$) is similar to the meaning of the relational equal sign ($=$) in the semiotic system of algebraic notation. This is not the case in Example 2, since in Scratch the equal sign ($=$) is always used as a relational operator and the double equal sign is not included in the syntax. We can summarize this as follows: In programming, the relational equality can be represented differently, for instance, as $==$ in JavaScript and as $=$ in Scratch, which means that a student needs to be able to convert between these two registers when learning to program. At the same time, when learning algebra, the equal sign ($=$) represents a relational equality, but never an assignment.

Furthermore, many instructions in programming, such as for and if, have built-in checks for truthfulness that closely correspond to the algebraic equality without including an equal sign in the syntax. An example of this can be seen at place 6 where the instruction `if(check)` should be understood as ‘if the variable *check* is equal to “true”, then ...’. Perhaps one might argue that in programming the relational equality sometimes is applied although it is hidden in the code, similar to the convention in algebra where multiplication is applied although not visible when we write $4 \cdot a$ as $4a$ (Hewitt, 2012).

We now return to the meaning of the variable concept in programming and algebra. First, in programming variables can be used to hold non-numbers such as in places 1 and 5 where the variable *check* holds the Boolean value “true” and then “false”. Second, in programming a variable can change value during the execution of the program. This is illustrated at the places 1, 3 and 5 in [Figure 3](#) and in the *sum* and *count* variables in Example 2. As already mentioned, at place 3 the variable a increases with 1 for every execution ($a = a + 1$). This means that if, for instance, a is equal to 3, the computer calculates the value $3 + 1$ and then a is assigned the new value 4, i.e. a changes value from 3 to 4. As we are about to see, this differs from the algebraic context.

In algebra, the concept of a variable is broad and has been further categorized as *unknown*, *variable* (in a more nuanced meaning) and *placeholder* (Ely & Adams, 2012). The term unknown corresponds to a determinate quantity in an equation that remains to be solved. Clearly, an unknown cannot change its value as it is predetermined from the equation. The term variable is meant to correspond to a varying quantity, typically x and y in the equation $y = x^2 + 1$. The variables in this equation may seem to change value, but the change is related to different cases of the problem. E.g. “Case 1: we assume x is 1, then y will be 2”, and then “Case 2: we assume x is 2, then y will be 5”. Within each of these cases we allow the variable x to have a determinate value which will make y into an unknown (that can be easily resolved). Observe that we cannot assign x the value 1 and later derive another value for x within the same case, that would constitute a contradiction ($x = 1$ and $x = 2$ imply $1 = 2$) and force us to eliminate this case. This is different from programming where instructions are executed in

order, one after the other. When an instruction assigns a new value to a variable x it does not constitute a new case, instead we are just changing the value recorded in a specific place in the memory of the machine (referred to by the variable). This could also be interpreted in terms of *time* in the sense that in programming variables are allowed to change value over time. In algebra there is no time aspect, although the same variable may be substituted for different values corresponding to different independent cases.

In summary, Example 3 highlights some differences in the meanings applied to the equal sign in the different registers and deepens the discussion about differences between the use and meaning of variables. Variables in programming registers can be non-numerical and can change values as an effect of time in the execution of a program, which is not the case in algebraic notation used in school algebra.

Discussion

In accordance with Malara and Navarra (2018), we suggest that, theoretically, both AT and CT address the structure of a process more than its result. However, as we saw in the example of physics described by Sherin (2001), a focus on process and causality invoked by a programming register differs from an understanding of balance and equilibrium, which are characteristics of algebra. Our analyses indicate that the mathematical structures that come to the fore in programming also give precedence to processes and time-related change, while downplaying the relational meaning of equality and inequality (see Examples 2 and 3). The differences with respect to the meaning of equality and inequality in algebra and programming could *afford* the development of students' algebraic thinking through contrasting examples and awareness of accuracy. However, it can also *constrain* it if the teacher is unaware of the different experiences students have. In particular, the equal sign in algebra is already a cause for much didactical effort in helping students switch from an operational to a relational meaning (Kieran, 1981; 2018). When programming introduces yet another meaning as well as a different syntax, this could cause misconceptions (Qian & Lehman, 2017) and have consequences for the learning of algebra.

We agree with Mason (2018) that computer languages could be used to invoke algebraic thinking and awareness. Even though programming languages originated from algebra, our examples illustrate that computer environments bring along new registers with syntactic rules that are different from those of natural language or of algebraic notation. This means that activities that may seem to invoke algebraic awareness could also divert students' attention away from algebra, for example, through emphasizing side effects (see Example 1) and variables that change value during an execution of a program (see Examples 2 and 3). In addition, the early introduction of non-numerical variables goes beyond school algebra where the numerical aspect of a variable is emphasized. In programming the variable changes value in the execution of a program, indicating and highlighting the process. In algebra variables vary when they describe a relation. The role of the variable is thus different in the two systems of representations – controlling the process and storing data in programming and expressing relations in algebra.

The fact that words have different meanings in different contexts shows that natural language is indeed vague and multi-functional, and perhaps is better understood here as two separate, but largely overlapping, registers. One register includes mathematical terminology and the other incorporates words and concepts specific to computer programming activities. The reason for separating them is that some words, such as algorithm and variable, are used in both registers but with different meanings, thus being similar representations referring to different objects. According to Duval (2006, p. 107), “[...] mathematical processing always involves substituting some semiotic representation for another”. Making a conversion from natural language to another register, by representing for example, “a variable” in a computer language or in algebraic notation, requires knowledge about what object the word variable refers to. For learners, the different meanings of a variable in algebra are already a challenge (cf. Bush & Karp, 2013; Ely & Adams, 2012; Usiskin,

1988), so the addition of yet another meaning related to computer programming is something that teachers need to be very explicit about.

Turning back to the overall aim of this paper, to explore how aspects of CT developed by programming are connected to AT, we will again consider the three content strands included in Kaput's (2008) definition of early algebra. We can conclude that CT developed through programming activities can be described in a similar way. It includes: i) a focus on structures, although these structures are more process-oriented and algorithmic than relational (Examples 1 and 2); ii) the study of functions, but these functions also have side effects and not always clear input- and output values (Examples 1 and 2); and iii) applies a cluster of modeling languages, with a diversity of syntactic rules and sometimes inconsistent semantic meanings (Examples 2 and 3). We argue that contemporary descriptions of algebra, in particular of early algebra (e.g., Kaput, 2008; Radford, 2018), may have become too general, blurring the differences between programming and algebra. We do not want to lose sight of algebra and the symbolic language developed over centuries because a new, and in many ways very useful, way of symbolizing and modeling has emerged. There is a risk that stakeholders promoting computational thinking may see AT as incorporated within CT. Just like algebraic notation, a programming language is a culturally evolved mode of symbolizing indeterminate quantities. But, these different registers do not have the same value and use when it comes to analytically dealing with quantities. We urge researchers to revisit current definitions in order to address what may be lost if algebraic notation is abandoned in favor of programming languages. We also call for more research about differences between algebraic thinking, algorithmic thinking and computational thinking. Undoubtedly, all three domains are essential when working with mathematics, so in order to teach them we need to see what each is specifically useful for.

Finally, we would like to point to the rather ironic turn of events when programming environments developed for other purposes than those of exploring mathematical ideas become the main source of inspiration for programming activities in mathematics classrooms. Should we pursue the route of investigating possible ways of using computer environments such as Scratch, simply because it is a way into computational thinking, or should we call for future programming environments specifically developed for mathematical explorations?

Notes

1. According to: <https://scratch.mit.edu/statistics/Scratch> had almost 52 million registered users and 51 million shared projects (13 March 2020).
2. <https://larportalen.skolverket.se/#/moduler/1-matematik/Grundskola/L%C3%A4rare%20i%20matematik>.
3. <https://lightbot.com/>.
4. <https://code.org/learn>.
5. <http://scratched.gse.harvard.edu>.

Acknowledgments

This work was supported by the Swedish Research Council [Grant no. 2018-03865]. We would like to thank Lennart Rolandsson, Uppsala University, and the thematic working group in algebraic thinking (TWG 3) at the CERME11 conference in Utrecht for valuable comments and suggestions.

Notes on contributors

Kajsa Bråting is Associate Professor at Department of Education, Uppsala University. Her research focuses on algebraic thinking, programming, history of mathematics and analyses of textbooks and curriculum documents. She leads the research project Integrating programming in school mathematics – exploring the intersection of algebraic and computational thinking funded by the Swedish Research Council. She has also participated in a research project aiming at characterizing the algebraic content in Swedish mathematics curricula and textbooks for primary and secondary school.

Cecilia Kilhamn, PhD, works at the National Center for Mathematics education (NCM) at Gothenburg University in Sweden. Besides her research, she works mainly with in-service teacher development. She held a postdoctoral position at Gothenburg University 2011-2014 and was a guest researcher at Uppsala University 2018-2019. Her research interests concern mathematics teaching and learning in primary and elementary school, with a specific focus on algebra, classroom communication and programming.

ORCID

Kajsa Bråting  <http://orcid.org/0000-0002-8169-5670>

Cecilia Kilhamn  <http://orcid.org/0000-0003-2294-4996>

References

- Aho, A. V. (2012). Computation and computational thinking. *The Computer Journal*, 55(7), 832–835. <https://doi.org/10.1093/comjnl/bxs074>
- Altadmri, A., & Brown, N. C. C. (2015). 37 Million compilations: Investigating novice programming mistakes in large-scale student data. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE'15)* (pp. 522–527). New York: ACM.
- Arcavi, A. (1994). Symbol sense: Informal sense-making in formal mathematics. *For the Learning of Mathematics*, 14(3), 24–35. <https://flm-journal.org/Articles/BFBFB3A8A2A03CF606513A05A22B.pdf>
- Bednarz, N., Kieran, C., & Lee, L. (1996). Approaches to algebra: Perspectives for research and teaching. In N. Bednarz, C. Kieran, & L. Lee (Eds.), *Approaches to algebra: Perspectives for research and teaching* (pp. 3–12). Kluwer.
- Benton, L., Hoyles, C., Kalas, I., & Noss, R. (2017). Bridging primary programming and mathematics: Some findings of design research in England. *Digital Experiences in Mathematics Education*, 3(2), 115–138. <https://doi.org/10.1007/s40751-017-0028-x>
- Blanton, M., Stephens, A., Knuth, E., Murphy Gardiner, A., Isler, I., & Kim, J. (2015). The development of children's algebraic thinking: The impact of a comprehensive early algebra intervention in third grade. *Journal for Research in Mathematics Education*, 46(1), 39–87. <https://doi.org/10.5951/jresmetheduc.46.1.0039>
- Bocconi, S., Chiocciariello, A., & Earp, J. (2018). *The Nordic approach to introducing computational thinking and programming in compulsory education*. Report prepared for the Nordic@BETT2018 Steering Group. <https://doi.org/10.17471/54007>
- Bråting, K., Kilhamn, C., & Rolandsson, L. (2020, January). Integrating programming in Swedish school mathematics: Description of a research project. *Presented at MADIF12, the twelfth research seminar of the Swedish society for research in mathematics education*, Linnaeus University, Växjö, Sweden.
- Brennan, K., & Resnick, M. (2012, April). New frameworks for studying and assessing the development of computational thinking. *Paper presented at the 2012 annual meeting of the American Educational Research Association*, Vancouver, Canada (pp. 1–25).
- Brousseau, G. (1997). *Theory of didactical situations in mathematics*. Kluwer.
- Brown, N., Sentance, S., Crick, T., & Humphreys, S. (2014). Restart: The resurgence of computer science in UK schools. *ACM Transactions on Computing Education*, 14(2), 1–22. doi:10.1145/2602484
- Bush, S., & Karp, K. (2013). Prerequisite algebra skills and associated misconceptions of middle grade students: A review. *The Journal of Mathematical Behaviour*, 32(3), 613–632. <https://doi.org/10.1016/j.jmathb.2013.07.002>
- Denning, P. J. (2017). Remaining Trouble Spots with Computational Thinking. *Communications of the ACM*, 60(6), 33–39. <https://doi.org/10.1145/2998438>
- Duval, R. (2006). A Cognitive Analysis of Problems of Comprehension in a Learning of Mathematics. *Educational Studies in Mathematics*, 61(1–2), 103–131. <https://doi.org/10.1007/s10649-006-0400-z>
- Ely, R., & Adams, A. E. (2012). Unknown, placeholder, or variable: What is x? *Mathematics Education Research Journal*, 24(1), 19–38. <https://doi.org/10.1007/s13394-011-0029-9>
- Falkner, K., Vivian, R., & Falkner, N. (2014). The Australian digital technologies curriculum: Challenge and opportunity. In J. Whalley & D. D'Souza (Eds.), *Proceedings of the Sixteenth Australasian Computing Education conference* (pp. 3–12). Auckland: Australian Computer Society.
- Fisher, L. (2016). A decade of ACM efforts contribute to computer science for all. *Communications of the ACM*, 59(4), 25–27. <https://doi.org/10.1145/2892740>
- Futschek, G. (2006, November). Algorithmic thinking: The key for understanding computer science. In *International conference on informatics in secondary schools-evolution and perspectives* (pp. 159–168). Berlin: Springer.
- Gazoni, R. M. (2018). A Semiotic Analysis of Programming Languages. *Journal of Computer and Communications*, 6(3), 91–101. <https://doi.org/10.4236/jcc.2018.63007>
- Grover & Pea. (2013). Computational Thinking in K–12: A Review of the State of the Field. *Educational Researcher*, 42(1), 38–43. <https://doi.org/10.3102/0013189X12463051>

- Hewitt, D. (2012). Young students learning formal algebraic notation and solving linear equations: Are commonly experienced difficulties avoidable? *Educational Studies in Mathematics*, 81(2), 139–159. <https://doi.org/10.1007/s10649-012-9394-x>
- Hickmott, D., Prieto-Rodriguez, E., & Holmes, K. (2018). A Scoping Review of Studies on Computational Thinking in K–12 Mathematics Classrooms. *Digital Experiences in Mathematics Education*, 4(1), 48–69. <https://doi.org/10.1007/s40751-017-0038-8>
- Holmes, K., Prieto-Rodriguez, E., Hickmott, D., & Berger, N. (2018, November). Using coding to teach mathematics: Results of a pilot project. *Integrated education for the real world. 5th international STEM in education conference* (pp. 21–23). Queensland University of Technology, Brisbane, Australia.
- Hoyles, C., & Noss, R. (2003). What can digital technologies take from and bring to research in mathematics education? In A. Bishop, M. K. Clements, C. Keitel-Kreidt, J. Kilpatrick, & F. K. S. Leung (Eds.), *Second international handbook of mathematics education* (pp. 323–349). Dordrecht.
- Kamii, C., & Dominick, A. (1997). To teach or not to teach algorithms. *Journal of Mathematical Behavior*, 16(1), 51–61. [https://doi.org/10.1016/S0732-3123\(97\)90007-9](https://doi.org/10.1016/S0732-3123(97)90007-9)
- Kaput, J. (2008). What is algebra? What is algebraic reasoning? In J. Kaput, D. Carraher, & M. Blanton (Eds.), *Algebra in the early grades* (pp. 5–18). Lawrence Erlbaum.
- Kieran, C. (1981). Concepts associated with the equality symbol. *Educational Studies in Mathematics*, 12(3), 317–326. <https://doi.org/10.1007/BF00311062>
- Kieran, C. (Ed.). (2018). *Teaching and learning algebraic thinking with 5-12-year-olds*. Springer.
- Kieran, C., Pang, J. S., Schifter, D., & Ng, S. F. (2016). Early Algebra: Research into its nature, its learning, its teaching. In G. Kaiser (Ed.), *ICME 13 Topical Surveys*. Springer Open.
- Kilhamn, C., & Bråting, K. (2019). *Algebraic thinking in the shadow of programming*. In U. T. Jankvist, M. van den Heuvel-panhuizen, & M. Veldhuis (Eds.), *Proceedings of the Eleventh Congress of the European Society for Research in Mathematics Education, CERME11* (pp. 566–573). Utrecht, the Netherlands: Freudenthal Group & Freudenthal Institute, Utrecht University and ERME.
- Kilhamn, C., Røj-Linberg, A.-S., & Björkqvist, O. (2019). School algebra. In C. Kilhamn & R. Säljö (Eds.), *Encountering algebra. A comparative study of classrooms in Finland, Norway, Sweden and the USA* (pp. 1–12). Springer.
- Kotsopoulos, D., Floyd, L., Khan, S., Namukasa, I. K., Somanath, S., Weber, J., & Yiu, C. (2017). A Pedagogical Framework for Computational Thinking. *Digital Experiences in Mathematics Education*, 3(2), 154–171. <https://doi.org/10.1007/s40751-017-0031-2>
- Malara, N. A., & Navarra, G. (2018). New words and concepts for early algebra teaching: Sharing with teachers' epistemological issues in early algebra to develop students' early algebraic thinking. In C. Kieran (Ed.), *Teaching and learning algebraic thinking with 5-to 12-year-olds* (pp. 51–77). Springer.
- Mannila, L., Dagieni, V., Demo, B., Grgurina, N., Mirolo, C., Rolandsson, L., & Settle, A. (2014). Computational thinking in K-9 education. In A. Clear & R. Lister (Eds.), *Proceedings of the Working Group Reports of the 2014 on Innovation & Technology in Computer Science Education Conference* (pp. 1–29). New York: ACM.
- Mason, J. (2018). How early is too early for thinking algebraically? In C. Kieran (Ed.), *Teaching and learning algebraic thinking with 5-to 12-year-olds* (pp. 3–25). Springer.
- Noss, R., & Hoyles, C. (1996). *Windows on mathematical meanings: Learning cultures and computers (Mathematics education library Vol. 17)*. Kluwer Academic Publishers.
- Papert, S. (1980). *Mindstorms: Children, computers, and powerful ideas*. Basic Books.
- Papert, S. (1996). An exploration in the space of mathematics educations. *International Journal of Computers for Mathematical Learning*, 1(1), 95–123. <https://doi.org/10.1007/BF00191473>
- Qian, Y., & Lehman, J. (2017). Students' misconceptions and other difficulties in introductory programming: A literature review. *ACM Transactions on Computing Education*, 18(1), 1–24. doi:10.1145/3077618
- Radford, L. (2018). The emergence of symbolic algebraic thinking in primary school. In C. Kieran (Ed.), *Teaching and learning algebraic thinking with 5-to 12-year-olds* (pp. 3–25). Springer.
- Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B., & Kafai, Y. (2009). Scratch: Programming for all. *Communications of the ACM*, 52(11), 60–67. <https://doi.org/10.1145/1592761.1592779>
- Sherin, B. L. (2001). A comparison of programming languages and algebraic notation as expressive languages for physics. *International Journal of Computers for Mathematical Learning*, 6(1), 1–61. <https://doi.org/10.1023/A:1011434026437>
- Sirkia, T., & Sorva, J. (2012). Exploring programming misconceptions: An analysis of student mistakes in visual program simulation exercises. In *Proceedings of the 12th Koli Calling International Conference on Computing Education Research (Koli Calling'12)* (pp. 19–28). New York: ACM.
- Spuler, D. A., & Sajeev, A. S. M. (1994, January). *Compiler Detection of Function Call Side Effects*. CiteSeerX. James Cook University. Retrieved 2019-06-24.
- Stephens, M. (2018). Embedding algorithmic thinking more clearly in the mathematic curriculum. *ICME 24 School mathematics curriculum reforms: challenges, changes and opportunities*. Tsukuba, 26-30 November 2018.

- Swedish National Agency for Education. (2017). *Läroplan för grundskolan, förskoleklassen och fritidshemmet, Lgr11* (National curriculum of Sweden). Fritzes.
- Usiskin, Z. (1988). Conceptions of school algebra and uses of variables. In A. Coxford & A. Shulte (Eds.), *The ideas of algebra, K-12. 1988 Yearbook* (pp. 8–19). National Council of Teachers of Mathematics.
- Wing, J. (2006). Computational thinking. *Communications of the ACM*, 49(3), 33–36. <https://doi.org/10.1145/1118178.1118215>
- Wu, M. L., & Richards, K. (2011). Facilitating Computational Thinking through Game Design. In M. Chang, W. Y. Hwang, M. P. Chen, & W. Müller (Eds.), *Edutainment Technologies. Educational Games and Virtual Reality/Augmented Reality Applications. Edutainment 2011. Lecture Notes in Computer Science* (Vol. 6872, pp. 220–227). Springer.