This is the published version of a paper presented at Eleventh Congress of the European Society for Research in Mathematics Education (CERME11), Utrecht, 6-10 February, 2019.

Citation for the original published paper:

Algebraic thinking in the shadow of programming

N.B. When citing this work, cite the original published paper.

Permanent link to this version:
http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-403224
Algebraic thinking in the shadow of programming

Cecilia Kilhamn and Kajsa Bråting

Uppsala University, Sweden; cecilia.kilhamn@edu.uu.se; kajsa.brating@edu.uu.se

This paper calls attention to how the recent introduction of programming in schools interacts with the teaching and learning of algebra. The intersection between definitions of computational thinking and algebraic thinking is examined, and an example of a program activity suggested for school mathematics is discussed in detail. We argue that students who are taught computer programming with the aim of developing computational thinking will approach algebra with preconceptions about algebraic concepts and symbols that could both afford and constrain the learning of algebra.

Keywords: Algebra, algebraic thinking, programming, computational thinking.

Introduction and background

In the wake of introducing programming into school mathematics curricula (Mannila et al., 2014) we have identified the intersection of algebraic thinking and computational thinking as an important area to explore (Figure 1). Specifically, this intersection has come to the fore in Sweden, where programming has been inserted into the national mathematics curriculum as part of the core content of algebra. Computational thinking (CT) is a fairly new concept in educational research, first introduced by Papert in 1996. The term involves the kind of thinking skills needed to understand and capitalize on computers. Since Wing launched CT as a didactical term in 2006, researchers in computational science as well as mathematics education have attempted to define it. For example, Hoyles and Noss (2015) describe CT as incorporating four central thinking skills: decomposition, pattern recognition, abstraction and algorithmic thinking. Programming, on the other hand, can be seen as a problem-solving activity that can be used to address the different aspects of CT (Manilla et al, 2014). Although CT is generally considered to encompass more than programming, teaching programming requires the use of CT (Hickmott, 2017). Moreover, programming is a feature of CT that does not necessarily involve writing code in any particular computer language (Bocconi, Chioccarello & Earp, 2018). Programming is thus a more inclusive term than coding, and seen as an activity in which students develop computational thinking. In this paper, we limit the discussion to aspects of CT that can be developed through programming. The aim of this paper is to raise the question of how programming in schools, with the goal of developing CT skills, may potentially interact with, afford or constrain students’ development of algebraic thinking.

Figure 1: The intersection of CT and AT

A contemporary international movement in school development is to implement programming in the curriculum. This has been done in various ways (Mannila et al., 2014). For instance, in England, “Computing” was introduced as a new subject in the curriculum, while Finland and Sweden adopted a blend of cross-curriculum and single subject integration with the strongest link to mathematics (Bocconi et al., 2018). Unlike other countries, Sweden included programming in the mathematics
curriculum in close connection to algebra through all grade levels. The curriculum change was released in August 2017 and was taken in effect throughout the country by August 2018. There is extensive literature from the last three decades tackling issues of teaching and learning programming at university level (Grover & Pea, 2013), but except for the early work on LOGO programming there are few studies of teaching programming in the early school years (Blikstein, 2018; Hickmott, Prieto-Rodriguez & Holmes, 2018). Furthermore, most educational research about programming concerns the learning of programming per se, rather than learning mathematical ideas through programming. One exception is the ScratchMaths research program, which in light of the recent curriculum changes in England, set out to explore the relationship between computational thinking and mathematical thinking in general, identifying mathematical ideas such as place value, proportional relationships, coordinate systems, symmetry and negative numbers in ScratchMath activities (Benton, Hoyles, Kalas, & Noss, 2017). We wish to broaden the discussion about “learning to program” or “programming to learn” by raising questions about how learning (and knowing) programming influences the learning of algebraic thinking. 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 in the wake of programming activities.

Programming activities in Swedish schools

The description of programming in the Swedish revised curricula and on-line material provided by the government, focuses in grades 1-3 on the use of symbols to construct and follow stepwise instructions either without computers (unplugged) or with simple robots such as Bee Bots or screen equivalents such as Lightbot. In grades 4-6 algorithms are created in visual programing environments such as Scratch. and in grades 7–9 other programming languages are introduced. In the visual environments, an algorithm is often described as a “recipe” or a “function”, and if variables are introduced, they are mostly non-numerical. While a function is described as a correspondence between two sets in abstract algebra, a function in school algebra is commonly interpreted as a co-variation of numerical values possible to represent in a graph (Usiskin, 1988). If students in the early grades meet functions as a correspondence between non-numerical input and output variables in programming activities, they will have experiences that may challenge the school algebra conception of function where one quantity is said to vary depending on another quantity, represented in a cartesian graph. Due to the limited space of this paper, we leave to the side further investigations of early programming activities in relation to algebra and focus our discussion on programming in text-based environments commonly included in later grades.

Algebraic thinking and programming

As algebraic thinking (AT) slowly made its way to the early school years during the last decades, the definition of what algebra is and what comprises algebraic teaching has become wider and more inclusive (Kieran, 2018). A broad definition of AT based on work with young children is suggested by Radford (2018) who describes AT as analytically dealing with indeterminate quantities using

1 https://larportalen.skolverket.se/#/moduler/1-matematik/Grundskola/Lärare%20i%20matematik
2 https://www.bee-bot.us; http://lightbot.com
3 http://scratched.gse.harvard.edu
culturally and historically evolved modes of symbolizing. Although this definition excludes operating on specific numbers, it includes generalized arithmetic such as analyzing properties of addition and multiplication or analyzing the structure of an algorithm (for example using the distributive property to work out the partial products in a multiplication algorithm). We set out to investigate if this definition of algebraic thinking excludes or includes aspects of computational thinking that are at the heart of programming. We note that in a contemporary state-of-the-art publication written by members of an ICME topic study group on the teaching and learning of early algebra at ICME 13 (Kieran, 2018) there is no discussion at all about the connections between the learning of algebra and computer programming. The word “programming” is never mentioned and “computer” appears only in one paragraph (out of 17 chapters), where Mason claims that computer languages have emerged from algebra, and that these “expressive and manipulable languages […] invoke algebraic thinking and algebraic awareness” (Mason, 2018, p. 335). Kaput (2008) describes early algebra in terms of three content strands including: i) the study of structures; ii) the study of functions; and iii) the application of a cluster of modelling languages both inside and outside of mathematics. In addition, symbolizing is described as core aspects of algebra (ibid). We find that all three content strands could also be aspects of computer programming. As an example, constructing code to generate all odd numbers presupposes reflection on what an odd number is and how it could be described and represented. Such an exercise is a study of structure, modelled in a specific programming language. Although the study of structure in algebra (e.g. generalized arithmetic) is emphasized in more recent work (e.g. Kieran, 2018), such structures have not been related to structures that becomes visible when breaking down complex calculations into small steps in a computer program. Although structure seems to be fundamental in both domains, little has been said in previous research literature about the study of structures embedded in computer algorithms in relation to the study of structure described as an aspect of algebraic thinking.

Investigating the intersection between computational and algebraic thinking

For the purpose of comparison, we have tried to illustrate similarities between two definitions of CT and two definitions of AT by aligning them in Table 1. The definitions are chosen as examples of contemporary definitions in the respective areas, proposed by researchers often cited in the literature. We see that both CT and AT deal with structure, generalization and symbolization in various ways, describing culturally developed organized symbol system where structures can be studied and generalizations made. Questions that arise are if structure, symbolization and generalization represent the same thing in the two domains, and if the syntax and semantics of programming languages are aligned with or divert from corresponding algebraic symbolism. Surprisingly, the definitions of CT do not explicitly mention functions or variables, which are fundamental concepts for AT (see Table 1). In programming activities, these concepts appear up front, where an algorithm in a program is often described as a function, and variables are omnipresent. Perhaps they are simply taken for granted in CT. Algorithmic thinking is not mentioned as part of AT, although it involves features that are fundamental in algebra, such as the ability to analyze a problem, specify it precisely, and find adequate basic actions (Futschek, 2006). Further, Futschek mentions the ability to construct a correct algorithm using the basic actions, to think about possible special and normal cases, and to improve the efficiency of an algorithm. We see such abilities as prominent when algebra is used as a problem-
solving tool where the basic actions are for example rules of equation solving or symbolic manipulation. Finally, according to Futschek, “algorithmic thinking has a strong creative aspect”, which undoubtedly should be true for algebraic thinking as well. Can we assume that students who develop their algorithmic thinking skills through programming also develop their algebraic thinking skills? Could algebra teaching explicitly relate to and build on algorithmic thinking? Note that algorithmic thinking in CT concerns the how and why of algorithms, as opposed to what Hiebert (2003) calls algorithmic reasoning, which is to recall a solution algorithm based on “rote thinking”.

Table 1: Some features of contemporary definitions of CT and AT that are found in the intersection

<table>
<thead>
<tr>
<th>Features in the intersection</th>
<th>Computational thinking (CT)</th>
<th>Algebraic thinking (AT)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hoyles &amp; Noss 2015</td>
<td>Symbol systems and representations</td>
<td>Using idiosyncratic or specific culturally and historically evolved modes of symbolizing</td>
</tr>
<tr>
<td>Radford 2018</td>
<td>Abstractions and pattern generalizations, Conditional logic.</td>
<td>Entailing abstraction</td>
</tr>
<tr>
<td>Kaput 2008</td>
<td>Functions and variables</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>Algorithms</td>
<td>Algorithmic notions of flow of control</td>
</tr>
</tbody>
</table>

* there is no mention of functions or variables in these definitions of CT
** there is no mention of algorithms in these definitions of AT

Next, we give an example of a programming activity in order to highlight some points of intersection with AT, in particular the issue of symbolization, taken from government-provided teaching materials available on line in connection to the revised curriculum in Sweden (Rolandsson, 2018).

An example of differences in syntax and semantics in algebra and programming

In this activity, the students are going to write a short computer program inspired by the algorithm of “Sieve of Erastothenes”, i.e., an algorithm for finding prime numbers. Examples are given in two programming languages: Python and Javascript. The latter is given in Figure 2 below. By using this algorithm, one can find out if a given integer is a prime number or not. We have highlighted six places in the code that illustrate some crucial passages regarding syntax and semantics. These places are circled and marked from 1-6 in Figure 2 and will be discussed next.

Let us begin to consider the equal sign. At the places 1, 2, 3 and 5 in Figure 2 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, i.e., 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 pupils tend to interpret the equal sign as an operator symbol \((5 + 3 = 8)\) rather than a relation (Kieran, 1981).
In algebra, it would be meaningless to write \( a = a + 1 \), since it is not true for any value of \( a \). However, in programming the expression \( a = a + 1 \) makes sense since the equal sign is understood as the assignment “add 1 to the value \( a \)” (see place 3 in Figure 2). This is often used when a program needs to loop through a range of consecutive integers, as in Figure 2 when the program loops from 2 to the input value stored in the variable “number”.

Let us now consider place 4 in Figure 2, 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, i.e. if the remainder is equal to 0 after dividing the input value “number” with the loop variable \( a \). Here, the double equal sign is used in a similar way as the relational equal sign in algebra.

Many instructions in programming, such as for and if, have built-in checks for truthfulness that closely corresponds to the algebraic equality without including an equal sign in the syntax. An example of this can be seen at place 6 in Figure 2 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 invisible in the code, similar to the convention in algebra where we write \( 4 \cdot a \) as \( 4a \) (Hewitt, 2012).

Let us now demonstrate some differences regarding 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 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 \(y = x^2 + 1\). It can be perceived that the variables in this equation can 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. The same argument can be applied for the term placeholder. 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).

**Discussion**

We have shown above how programming activities can interact with students’ development of algebraic thinking in different ways. One of these concerns symbolization. In algebraic thinking, the use of culturally and historically evolved modes of symbolizing (Radford, 2018) is described as a prominent feature. Notice the plural, indicating that algebraic thinking is not exclusively bound to alphanumerical symbols and conventional algebraic syntax, and could thus hypothetically incorporate also programming languages. While many programming languages use similar, but not equivalent, symbol systems, there is no common syntax. For example, in some languages the short hand notation \(a++\) (the increment operator) is used instead of \(a = a + 1\). Nevertheless, the differences between the established algebraic symbol system and new programming languages could potentially entail difficulties for students learning algebra.

In order to clarify some of these differences we have, from the example in Figure 2, compiled three categories regarding the meaning of some operations (the semantics) and the symbolic representations of these operations (the syntax) that appear in algebra and programming: 1) Different symbols represent the same meaning: e.g., the modulus operation, represented by \(mod\) in algebra and \(\%\) in programming; 2) The same symbol represents different meanings: e.g., the equal sign (=) representing relational equality in algebra and assignment in programming; 3) Symbols with no corresponding meaning in the two domains: e.g., \(\approx\) in algebra and increment with one (++) in programming. These differences could afford the development of algebraic thinking through contrasting examples and awareness of accuracy, or 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, this could cause more confusion. In contrast, the use of different symbols to represent the same meaning could help students understand that the meaning of a mathematical concept does not depend on the symbol used to represent it.

The meaning of variable is another issue found in the intersection of CT and AT. The use of non-numerical variables (such as true/false) in programming affords a wider understanding of the concept of variable than the traditional use of variables as quantities, and does not correspond to the school algebra idea that letters always represent numbers.
The focus on algorithms and operational execution of instructions in programming may interact with the focus on structure in algebra. As shown in Table 1, both CT and AT value the study of structure, decomposition and pattern recognition. Approaching questions about structure from two perspectives could serve as an affordance for developing AT if differences in structure are noticed, contrasted and discussed. Algorithms are important in programming but absent in descriptions of AT. In school mathematics, the idea and use of algorithms have changed greatly since the introduction of digital tools in the 1980’s when traditional algorithms were replaced by an increased emphasis on number sense and conceptual understanding. The fact that the term algorithm was removed from the description of arithmetic in the Swedish national curriculum in 2011 and re-inserted in 2017 as an aspect of the core content of algebra in connection to programming, implies a shift of emphasis from a procedural use of algorithms, to a conceptual understanding in terms of “algorithmic thinking” (Futschek, 2006). An open question for further empirical research is, if this newly awoken interest in algorithms will induce a study of the structures embedded in algorithms, and as such afford the development of algebraic thinking in terms of increased focus on structure and generalizations, or if it will tip over towards a more procedural approach to mathematics as a whole, thus working as a constraint to AT.

We argue that future algebra students who are taught programming in school, will encounter working with variables, structure and patterns, and engage in simplifying, generalizing and symbolizing in computer related contexts. We therefore urge for an awareness of what these experiences might do to how algebra is conceptualized and talked about in school, and what preconceptions teachers need to be aware of. We trust that empirical research into the intersection of AT and CT, as programming evolves in school practices, will highlight possible pitfalls and supply algebra teaching with new input.

References


