ZDM – Mathematics Education (2022) 54:1273–1288
https://doi.org/10.1007/s11858-022-01384-0

ORIGINAL PAPER

Variables in early algebra: exploring didactic potentials in programming activities

C. Kilhamn[1] · K. Bråting[2] · O. Helenius[1] · J. Mason[3,4]

Accepted: 11 May 2022 / Published online: 8 June 2022

Abstract
In this paper we consider implications of the current world-wide inclusion of computational thinking in relation to children's development of algebraic thinking. Little is known about how newly developed visual programming environments such as Scratch could enhance early algebra learning. The study is based on examples of programming activities used by mathematics teachers in Sweden, teaching students aged 10–12 years during the first two years of implementing programming in the mathematics curriculum. Informed by Chevallard's praxeology in terms of praxis and logos, we describe, unpack, discuss and expand these activities. Core issues related to algebra found in the three activities are as follows: making implicit variables explicit; using a counter variable; and identifying parameters as a specific type of variable. Our findings show that, in addition to already identified uses of variables in early algebra, programming activities in the early years bring in new aspects and new ways of treating variables that could, potentially, enhance students' understanding of variables and generalization, provided that programming praxis is embedded in an appropriate algebra logos.

Keywords Variables · Early algebra · Programming · Scratch · Logos · Praxis

1 Introduction

Over the past 50 years, school mathematics has transformed in many ways, and significantly so for the age group of 5- to 12-year-olds. One change is that algebra has moved down in the school system; early algebraic thinking has been introduced as an important part of the mathematics curriculum in primary school, and as a growing field of research (Kieran, 2018). In recent years, computational thinking has commenced on a similar trajectory, as programming is making its way into schools worldwide (Mannila et al., 2014). Programming activities, unplugged and as coding in a large variety of computer languages, have been introduced both as a tool for mathematics and as content in its own right. In many countries the connection to mathematics is emphasised by including programming in the mathematics curriculum (Bocconi et al., 2018), and in Sweden a specific connection has been made to algebra (Bråting et al., 2021). Regardless of the initial intentions and the path taken, once a digital tool has entered into the mathematics classroom it contributes to a transformation of school mathematics.

In the ICME-13 topical survey on Early Algebra (Kieran et al., 2016), the need for exploratory studies of digital tools in early algebra was identified. In this paper we seek to investigate how visual programming could change or contribute to the teaching and learning of variables in early algebra. The aim is to unpack different meanings of variables that come into play in programming activities. A focus on variables reveals a chasm between programming syntax and algebraic thinking, in which variables are used in slightly different ways, with sometimes inconsistent meanings (Bråting & Kilhamn, 2021; Partanen & Tolvanen, 2019). Here, we dig more deeply into these meanings, exploring what impact children's encounter with variables through programming could have on their development of algebraic thinking. The paper is based on previously collected teacher-designed activities in which programming was introduced in mathematics lessons. By analysing the teachers' lesson plans and evaluation of the classroom activities, and then tinkering with and redesigning the tasks used, we endeavoured to

✉ C. Kilhamn
cecilia.kilhamn@ped.gu.se

[1] University of Gothenburg, Gothenburg, Sweden

[2] Uppsala University, Uppsala, Sweden

[3] Open University, Milton Keynes, UK

[4] Oxford University, Oxford, UK

answer the following research questions: What aspects of variables appear in programming activities using Scratch? How can these activities be modified in order to exploit their didactic potentials in relation to early algebra?

2 Variables in early algebra

Using and manipulating variables represented by letters has been seen as a core element in school algebra. However, as the introduction of algebra in school mathematics has moved down the school system from traditionally being introduced in lower secondary school, algebraic thinking and the use of algebraic tools has been extended to incorporate much more than manipulation of alpha-numerical symbols (Kieran et al., 2016). The teaching and learning of algebra in the early grades can be approached from different perspectives, applying a variety of mathematical tools and representations. Some claim that algebraic thinking precedes understanding of arithmetic (Britt & Irwin, 2011) and research has shown that children can learn to handle symbols for as-yet-unspecified quantities even before they handle specific numbers (Davydov, 1972/1990). Others emphasise early algebra as generalised arithmetic or functional thinking (Blanton et al., 2011; Kaput, 2008). In an attempt to define algebraic thinking in the early grades, Kieran (2004) describes it as involving "ways of thinking within activities for which letter-symbolic algebra can be used as a tool but which are not exclusive to algebra and which could be engaged in without using any letter-symbolic algebra at all" (p. 149).

2.1 Different meanings of variables

Although the essence of algebra is about making claims about and operating on quantities that are not specified, the ability to operate on variables without evaluating them seems to be a struggle for students. Many researchers, following Küchemann (1978), have studied students' conceptions and misconceptions of variables (see Bush & Karp, 2013, for an overview). Traditionally, these conceptions were tied to the use of alpha-numerical letters, but in early algebra more unconventional and often transient symbol systems appear. Radford (2014) described embodied forms of non-symbolic algebraic thinking as a first step towards the use of culturally evolved symbol systems. Furthermore, early algebra researchers all highlight the essential role of natural language in the development of algebraic thinking (Kieran et al., 2016).

The use of variables is seen as fundamental for algebra (Carraher & Schlieman, 2007). In the early days of the Dutch *realistic mathematics education* movement, Treffers (1987) describes different aspects of variables and the representation of numbers by letters as an increasing level of abstraction within mathematics itself. He brought up four aspects of such representations, namely, *generality*—as any number, *variability*—that it may take different values, *specificity*—as a particular as-yet-unknown number, and *constancy*—a number that is unknown but stays the same. Over the years, many mathematics education researchers have described different meanings of variables in school algebra, identifying the same four aspects using slightly different terms, such as *generalised numbers*, *varying variables*, *unknowns,* and *placeholders* (e.g., Ely & Adams, 2012; Partanen & Tolvanen, 2019; Usiskin, 1988). Since letters are not always present in early algebra, Radford (2014) introduced the term *indeterminate quantities,* with the idea that children could meet generalization and structure through any kind of symbolic placeholders that represent as-yet-unknown or as-yet-unspecified quantities. Integrating the different interpretations of variables and giving them meaning is a considerable didactical challenge.

Drawing on several decades of research about algebraic thinking, Radford (2014) suggested three conditions that characterise early algebraic thinking, as follows: (a) *indeterminacy*, i.e., involving unknown numbers, variables, parameters, etc.; (b) *denotation*—the indeterminate numbers are named or symbolized in various ways; and (c) *analyticity*—the indeterminate quantities are treated as if they were known numbers. In this paper we adhere to these conditions to describe the use of variables in early algebra, also recognising that algebraic thinking is to a large extent the recognition and articulation of generality, of seeing the general through the particular and of seeing the particular in the general (Mason et al., 2005).

2.2 Variables in programming

In addition to semiotic tools and symbolism, the last few decades have produced an impressive development of digital environments to mediate mathematical learning. However, in the early years of computer development, Sutherland (1993) found that students' unassisted use of computer variables was strongly related to their first assisted use of the idea, thus highlighting the role of the teacher when digital tools are used. In recent years, the use and meaning of variables in programming have come to play an increasing role in mathematics, and highlighted a new domain for mathematics teachers to embrace.

Programming in school mathematics was first introduced during the 1980's with the development of the Logo environment, specifically designed to help students develop a mathematical cast of mind (Noss, 1986; Papert, 1980). Noss (1986) studied how school children of age 8 to 11 years learned about variables through logo programming and concluded that "children may—under the appropriate conditions—make use of the algebra they have used in a Logo

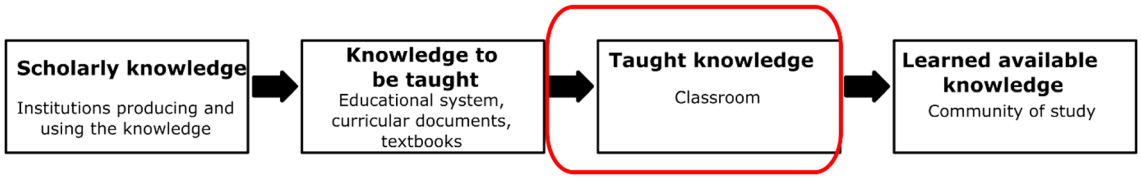

Fig. 1 The didactic transposition (Bosch & Gascón, 2006)

environment, in order to construct algebraic meaning in a non-computational context" (p. 354), but he also pointed out the necessity of linking experiences from Logo to the formalism of traditional algebra.

In programming, a variable can be described as a memory location, and the value of the variable as the contents of this memory register (Usiskin, 1988). The value assigned to the variable in the code is stored in the computer's memory and may be changed many times as the code is executed. This variability differs from school algebra, in which a variable always represents a number that stays the same throughout the whole algebraic expression and any manipulations made with it. In addition, an as-yet-unspecified number can often be determined at the end of an algebraic manipulation. In contrast to algebra, where we use single letters for the sake of easing manipulations, a computer can deal with long variable names. Naming variables with descriptive words has been shown to decrease errors, but moving from long variable names to short abbreviations or single letters can sometimes be a difficult transition for programmers (Hofmeister et al., 2019).

Compared to what we have been used to in school algebra, variables in early programming activities show a slightly different face (Bråting & Kilhamn, 2021). Firstly, they do not always represent quantities, and secondly, they are represented using different registers in which they can behave in idiosyncratic ways. Defining a variable x, and then writing, for example $x = x + 2$ is an untrue statement in algebra, but a perfectly reasonable instruction of assignment in a computer code. This example illustrates Duval's (2006) claim that signs make sense only within their own registers. At the same time, Duval argues, it is through transformations within and between registers that learners have access to abstract mathematical objects.

3 Theoretical framework

This study is part of a more comprehensive research project regarding the implementation of programming in Swedish school mathematics (Bråting et al., 2021). The project as a whole is embedded in Chevallard's (2006) framework of how knowledge is transposed between different instances of the educational system. The outcomes of the process of teaching and learning depend on the humans involved in it, starting with scholars developing and determining the knowledge at one end, and ending with the students' learned knowledge at the other (Fig. 1). In terms of this project, scholarly knowledge of programming as a science and as a tool for mathematicians is turned into knowledge to be taught and learned in mathematics classrooms. In each transposition, knowledge changes through interpretations and choices made by the individuals involved. Specifically, in this paper, we zoom in on the transposition of *knowledge of programming in mathematics* from 'Knowledge to be taught' to 'Taught knowledge' (encircled in red in Fig. 1).

In Chevallard's discourse of *transposition of knowledge,* presented tasks and activities evoke elements of *praxis* often referred to as 'know-how', including knowing-what and knowing-how. Allied with praxis is *logos*—the theoretical underpinnings and justifications of praxis, which guide teachers' didactical actions and their expectations of learners. The combination of logos and praxis constitutes what Chevallard (2006) calls a *praxeology*. Logos describes knowing-why certain tasks and techniques are introduced or employed. Depending on the relation between praxis and logos, a task can induce quite different kinds of learning.

In this paper, we analyse three programming tasks taken from authentic mathematics lessons and discuss how a transformation of logos could change the learning potential of a given task or technique. In the discourse of didactic transposition, a discussion of affordances and possible improvements of a task presumes a logos perspective, since the discourse in which a task makes sense co-determines what we consider as worthwhile learning from it. We look at praxis in light of the underlying logos. If praxis is justified by and centred around mainly computational thinking goals, programming syntax and variables in the computer sense of assignment, we identify the logos as CT (see Sect. 4). In contrast, an algebra logos implies algebraic learning goals and justifications in line with Radford's (2014) three conditions described above and Mason et al.'s (2005) emphasis on generalisation.

The sequence of didactic transpositions poses a number of issues. Allied with transpositions are the range of didactic choices available to teachers, and those that are enacted. A narrow range of available didactic actions necessarily leads to a narrowly determined learning pathway; a broad range

of available actions opens up the possibility of responsive and sensitive choices, enabling a richer learning pathway. Didactic choices are centred around what, in the moment, forms the object(s) of learning as perceived by the teacher. For example, a focus on variable as used in algebra is likely to bring different actions to the surface than does a focus on programming syntax.

As with any mathematical problem, *looking back*, as George Pólya (1954) expressed it, involves asking yourself how something could be used in the future, and tinkering with aspects of it in order to appreciate the scope of generality available from the one example. Re-flecting includes pro-flecting, imagining yourself in some future situation making use of what has been learned or encountered currently (Mason, 2002). Tinkering with examples, and noting not just the results of tinkering but the nature of that tinkering, enriches the space of examples to which you may have access in the future (Watson & Mason, 2005). Taking this to heart, in this paper we attempt to tinker with tasks presented by teachers, to see what they could potentially enhance in relation to algebra learning.

4 Programming in school mathematics

Recent years have seen a surge in introducing programming in school curricula (Blikstein, 2018). To a large extent this is driven by arguments related to preparing students for a digital society. In conjunction with this and as a complement to mathematical thinking skills, different ways of thinking about digital competences have been launched and relaunched, most importantly computational thinking (CT). Introduced by Papert (1980), the term was popularized again by Wing (2006) and later filled with detailed content in the form of categories of competencies (Brennan & Resnick, 2012).

4.1 Programming for children

In many countries we now find young children engaged in reasoning about structures and patterns in computer algorithms while creating, remixing and debugging code during mathematics lessons. A tradition that goes back to Papert, and beyond, is to build particular programming environments where children can explore the world of programming and develop computational thinking. The classic example is Logo (Papert, 1980) and the most prominent modern example is Scratch. While Logo to a large degree was dedicated to the learning of mathematics in a programming environment (Noss, 1986), the main goal of the Scratch environment is to increase digital fluency among children at large, by engaging them in creative digital design (Resnick et al., 2009). According to the website,[1] Scratch is a free-of-charge worldwide coding community for children, used by more than 200 million during 2021. It was launched in 2013 and is now available in 70 languages. Although popular and inspiring, Scratch does not automatically lead to mathematics learning. In the ScratchMaths project in the UK (Benton et al., 2017; Hoyles & Noss, 2015), design principles for programming activities in Scratch that also enhance mathematics learning were developed during an extensive teaching experiment. An independent evaluation of the project found that children in ScratchMaths schools made progress in CT, but there was no evidence of any impact on their mathematics outcomes (Boylan et al., 2018).

4.2 Programming and algebra—the Swedish case

The recently revised Swedish curriculum for mathematics (Swedish National Agency of Education, 2018) calls for the introduction of programming to be included in mathematics. Furthermore, it is placed within the core content of algebra, hence the inclusion of programming implies a change in how students encounter algebraic ideas such as variables and generalization. This radical inclusion gives us a unique opportunity to explore the intersection of programming and school algebra.

When the core content of algebra is described in the curriculum, programming is included as one among several other algebraic ideas, emphasising stepwise instructions and algorithms, first in visual programming environments and later in text-based environments. Other algebraic ideas are introduced gradually, as follows: equalities and patterns in grades 1–3; unknown numbers, algebraic expressions and equations, as well as the coordinate plane and simple graphs in grades 4–6; variables, linear equations, graphs and formulas as well as functions in grades 7–9. In all grades, different aspects of indeterminate quantities appear, with an increasing level of abstraction starting with unknown numbers.

When it comes to programming, which is new to all and briefly described, teachers are faced with the challenge of identifying learning objectives, planning activities, creating tasks, and choosing techniques that can be justified within what they recognise as algebra, or at least as mathematics. An analysis of the way programming was introduced in the Swedish curriculum describes it as praxis-oriented and almost 'logos free', leaving teachers with little guidance as to the purpose of teaching programming or the justification of choices they make (Helenius & Misfeldt, 2021).

In an effort to support teachers, the research institute Ifous[2] ran a three-year project with teachers using a lesson

[1] https://www.scratchfoundation.org/our-story (20 December, 2021).

[2] https://www.ifous.se/about/.

study design to develop lessons including programming, in line with the new curriculum. The final evaluation and project report (Jahnke, 2020), claimed that many of the tasks could potentially help develop CT competencies as described by Brennan and Resnick (2012). However, the report also showed that in the cases where mathematical learning was gauged, students in control groups using pen and paper methods learned more about the mathematical concepts and methods involved. Through interviews it was concluded that teachers struggled to use programming for mathematical learning. Learning to program seemed to be given priority over questions about how programming could be useful in relation to learning mathematics (Kilhamn et al., 2021).

In summary, the results from the Ifous project reinforced that it is hard to design mathematics lessons using Scratch code while also catering for mathematical development on a profound level. This is in line with the call from Gadanidis et al. (2017) that 'We need many more cases of what might be in mathematics and CT integration to better understand the role CT affordances might play in disrupting and improving mathematics education' (p. 94). In the current paper we report an effort to do so through using a selection of lessons and associated Scratch code from the Ifous project, and a follow-up project modifying them, to show how important mathematical learning opportunities related to algebra could be opened up.

5 The study

In this paper we present an in-depth analysis of three tasks taken from authentic mathematics lessons developed within the Ifous project mentioned above (Jahnke, 2020). In the project, 135 teachers were engaged in traditional lesson studies as described by Takahashi and Yoshida (2004). Their challenge was to develop lessons to teach programming within the frames of existing school subjects, when it was first introduced in the curriculum. In total, the project produced 32 mathematics lessons intricately balancing, on the one hand, learning of some traditional mathematics, and on the other hand, learning to use a specific programming environment. Each lesson was jointly planned by a group of teachers, who tried out, observed, evaluated and improved the lesson at least once in a traditional lesson study cycle. To report their work and share ideas within the larger group, the teachers created written documentations of their lesson studies. These documents were analysed earlier (Kilhamn et al., 2021), where findings showed that CT skills were foregrounded in most lessons, and that mathematics was often either non-existent (in 1/3 of the lessons) or serving only as a backdrop (in 1/3 of the lessons). Concerning mathematical content, the lessons were dominated by geometry and basic arithmetic. In particular, explicit algebra learning goals were absent. However, amongst the 32 mathematics lessons we found three, where variables were in play in different, but mathematically relevant, ways. The teachers' written documents describing the task, complemented by their written lesson plans and evaluations of each of these three lessons constitute the data for the current study. The tasks are seen as examples of authentic teacher-planned activities that could potentially enhance algebraic thinking.

All three tasks included code in the visual environment called Scratch, where a code is built by dragging pre-defined blocks from one area to another on the screen and then ordering them sequentially in stacks connecting them to each other like a jigsaw puzzle (Resnick et al., 2009). There are different types of blocks identified by colour, where for example variable blocks are orange and motion blocks are blue (see Fig. 3). The environment includes three interface areas, one with available blocks, one where the code is built, and one where things happen when the program is executed. Figures or icons in the latter area that are acted on through the program are called sprites.

Each of the three tasks was individually analysed by at least two of the authors with the aim of identifying praxis (what and how) and logos (arguments and justifications for the choice of what and how, including learning goals) in the way the task was enacted, specifically focussing on the use of variables. All tasks provided both unexploited potentials and constraints related to algebra. In several rounds of tinkering and joint reflection, we then designed extensions and modifications to exploit didactical potential of the tasks. By viewing the task from a logos perspective, we asked what aspects of variables it could potentially enhance, and then modified the task in line with an algebra logos.

In the following Sects. 6, 7 and 8, the three tasks *Drawing circles*, *Using a counter variable* and *A function machine* are presented, each in three parts. Part 1 poses the initial task as described by the teacher and presented in class. Part 2 describes our analyses of core issues related to the task from an algebra point of view, including a discussion of the teachers' intentions. In part 3 we tinker with the task to develop variations and extensions that could further enhance algebraic ideas and possibilities of generalisation, based on an algebra logos aligned with the definition of early algebra described by Radford (2014) and a focus on generality described by Mason et al. (2005).

6 Drawing circles

The first lesson was designed by three teachers and tried out first in grade 6 (with students of age 12 years) and then in grade 4 (age 10). According to the teachers, the aim of the lesson was to construct a geometrical object using a digital

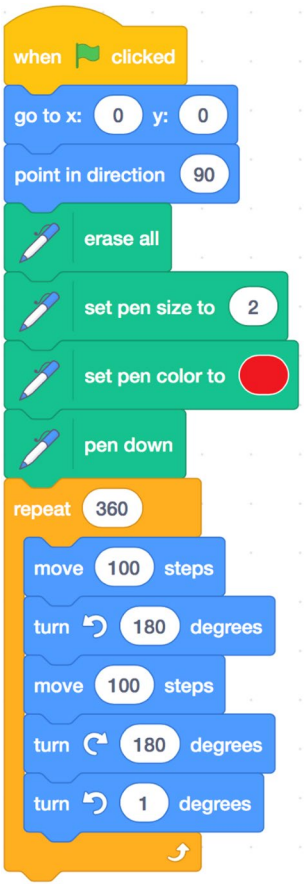

Fig. 2 Scratch code to draw a red circle

tool, but also to experience the relationship between the number of degrees in a circle and the number of degrees in a semicircle, and later in smaller parts of a circle. However, the learning objectives were, using the words of the teachers, to 'follow instructions, understand concepts, notice relationships, share knowledge and to code using Scratch'.

6.1 Initial task

At the start of the lesson, the teacher gave explicit step-by-step instructions showing how the Scratch code works (Fig. 2), expecting the students to construct the same code on their own computers. The teacher ran the program and discussed with the class why there is a repeat of 360 and a turn of one degree each time and why the result is a circle. Then the teacher asked: 'What do we need to change in the code to make a semicircle, a quarter of a circle or a third of a circle? Try it out! Help each other'. Eventually the final problem was posed, as follows: to make a full circle where half the surface is red and half is blue. The drawn figure was labelled a circle, with no distinction made between the disc and the boundary.

Table 1 Implicit variables in the original code

Implicit variable	Block
sprite_position	go to, move
sprite_rotation	point in direction, turn
pen_size	set pen size to
pen_color	set pen color to
pen_location	pen up, pen down
loop_counter	Repeat

The code starts by denoting a position (x, y) that will be the centre of the circle; adds direction with 90° being to the right on the screen; and sets pen size and colour. Then a loop is introduced where the pen draws lines from the centre to the boundary and back, turns 1° and repeats 360 times to create a full coloured disc.

6.2 Core issue: making implicit variables explicit

The original code does not include any variable blocks (orange blocks in Scratch), and there are no explicit variables in the standard mathematical sense. From a mathematical point of view, the current position of the sprite is its x- and y-coordinates, but these are kept implicit and never operated on directly except when they are both set to 0 at the start. Instead, because the environment provides several movement commands (*move* and *turn* are blue motion blocks, *set* and *pen up* are green operator blocks), the position and the associated x- and y-values are, so to say, 'operated on indirectly'. In fact, the program contains several such implicit variables, some presented in Table 1.

If we relate the technique used to a CT logos, the task seems rather successful, boosting computational concepts and practices as described by Brennan and Resnick (2012). The modification of making two semicircles of different colour decomposes the loop and the role of the number 360. But since the teachers do not mention variables in their lesson plan or evaluation, this aspect of the task can be seen as a hidden potential.

Several ways of modifying the task to enhance the learning of variables spring to mind, but choice of direction to explore depends on which kind of logos is foremost. Setting the task in an algebra logos would highlight the role of variables. Most of the implicit variables in Table 1 could be made explicit in the sense that they could be given a name and then be operated on. Once we have them as variables it is possible, (a) to let the program ask for values for these variables at the start of each run, and (b) to change or let the program change values and repeat the code. Doing so, would create indeterminate numbers that are symbolized and acted upon analytically.

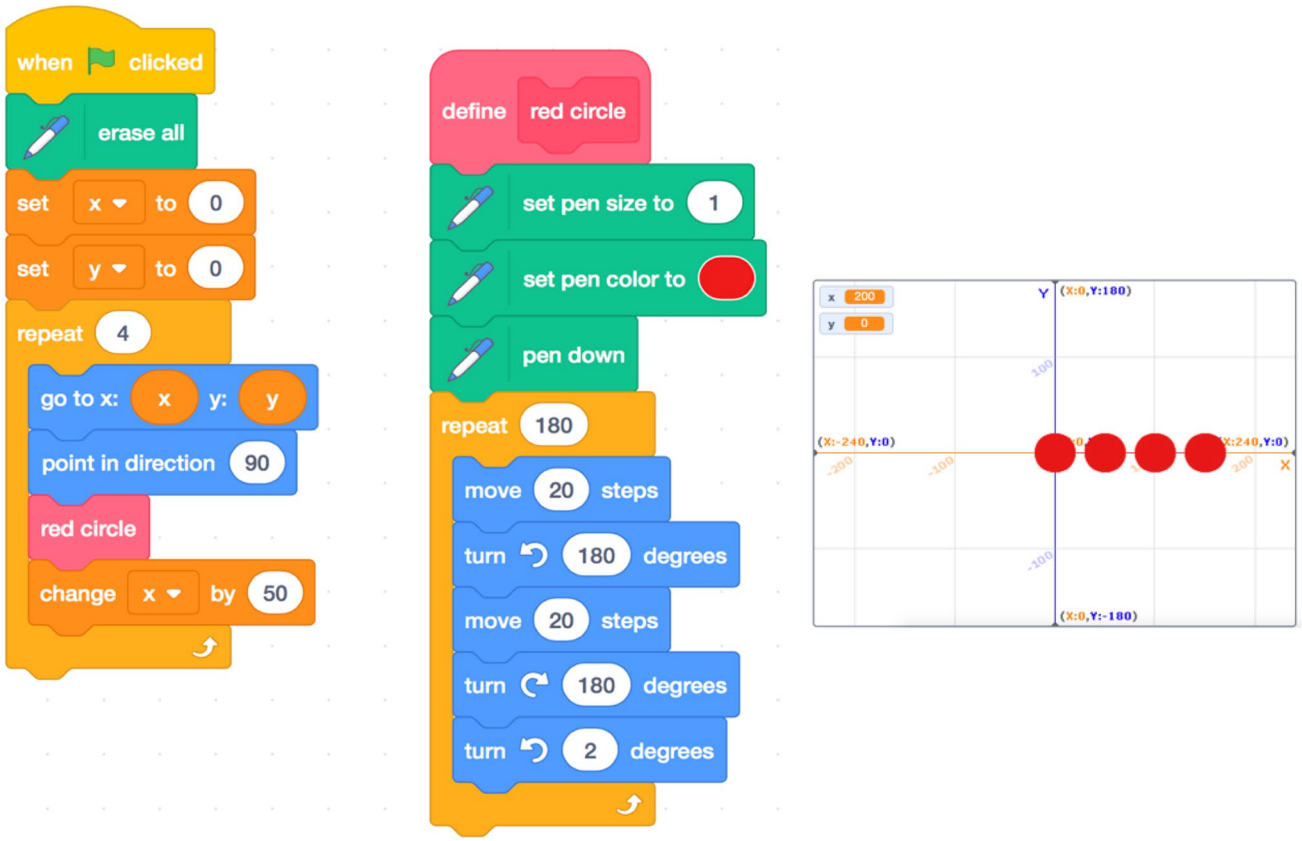

Fig. 3 Scratch code to draw four red circles. The left code uses the variable blocks 'set *x* to_' and 'set *y* to _', and then repeats the red circle described in the new block 'red circle'. For each repeat the *x* value is changed by 50. To the right is the result shown on screen when the code is run

6.3 Task modifications and possible didactic actions

Let us first consider the position of the sprite. At any state in which the program can be, the position is specified by the two quantities, *x* and *y*, which remain implicit. To make them explicit we can create variable blocks labelled *x* and *y* (see Fig. 3). To make the program change values, we defined a new red block that draws a red circle, and then placed it in a loop and added the variable block 'change *x* by 50'. This way the program draws several circles in succession. When this is done, we get nested loops and it is vital to understand what needs to be inside each loop and what remains outside. The code quickly becomes very long, which is a constraint when using Scratch that can be dealt with by defining new blocks. The outer loop now includes a block named 'red circle' which, in turn, includes a loop. By changing the background to a coordinate plane, it is also easier to locate the circle in relation to the *x* and *y* values, thus making the mathematics more visible. Figure 3 shows the code and the result when the program is run. Now it is possible to discuss the value of *x* and *y* in a more general sense and explore the variables *x* and *y* through inserting different starting values and changing values.

Another extension would be to set the task in a computer graphics context and ask how well the program draws a circle (or disc). The implicit variables that affect the resulting image are pen size, number of rays, and number of steps moved. To assist exploration, we made them explicit by choosing orange variable blocks in the Scratch code and naming them *raythickness*, *raynumber* and *radius* (see Fig. 4).

Results of a selection of values on these variables is shown in Fig. 5. The original choice in Fig. 4 produces a reasonable image of a disc, see left part of Fig. 5. To the right is the result when only drawing 90 rays. What are appropriate choices for the number and thickness of the rays for a given radius? The teacher can let the students experiment with different combinations and come up with hypotheses on suitable combinations of values. For example, few but thick rays will cover the disc, but produce a 'flowery' perimeter. Using a very high number of rays will make the drawing slow. At some point, the teacher could ask if there is a general rule that produces a reasonable image. One insight gained by

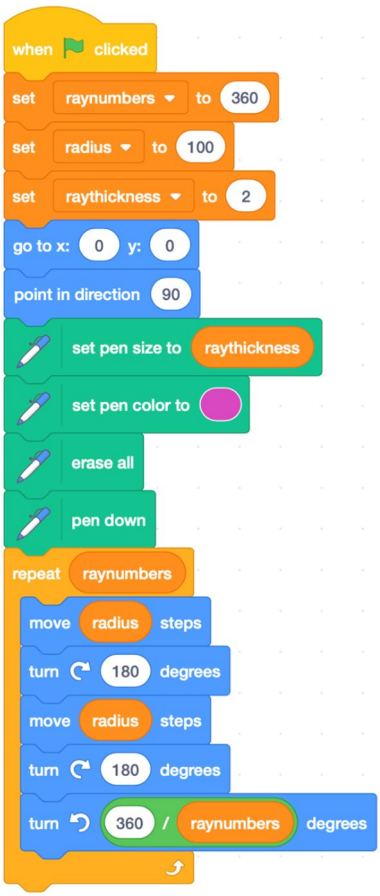

Fig. 4 Scratch code in which implicit variables have been made explicit

exploration could be that a minimum requirement to avoid fraying is that the accumulated width of all the rays must be at least as large as the circumference of the circle.

7 Using a counter variable

The second lesson was designed by three teachers, aimed at students in grade 6 (age 12) and also tried out with programming novices in grades 7 and 8. According to the teachers, the learning objective was 'to understand how programming can help create algorithms to solve and visualise mathematical methods'. The mathematical object of learning was to enact a particular multiplication by using repeated addition and to 'get an introduction to new mathematical concepts, for example variable'. The teachers reported that the students already had a good understanding of multiplication as repeated addition, and they found that programming syntax became the focus of attention.

7.1 Initial task

The teacher started the lesson by writing on the board: $3+3+3+3+3$, and asking whether it is possible to simplify, leading on to 'what does multiplication mean?'. Instructions were then put on the board:

Copy the Scratch code (Fig. 6)
Run the program
Try to compress the code
Try changing the operation

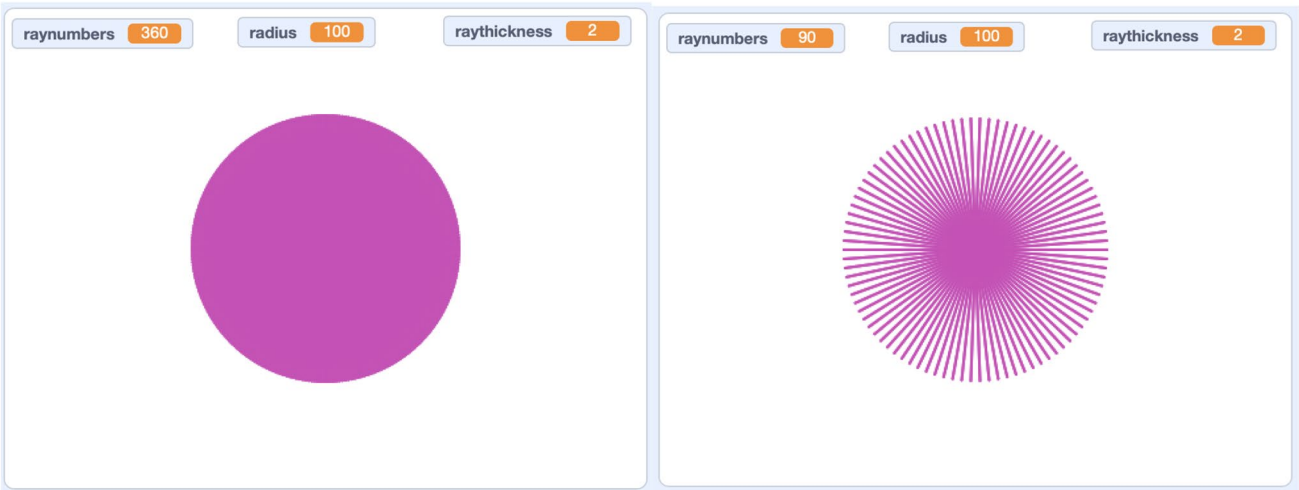

Fig. 5 At radius 100 and ray thickness 2, 360 rays (left image) will produce a shape of a disc while 90 rays (right image) will produce an image with frayed edges where the individual rays are visible

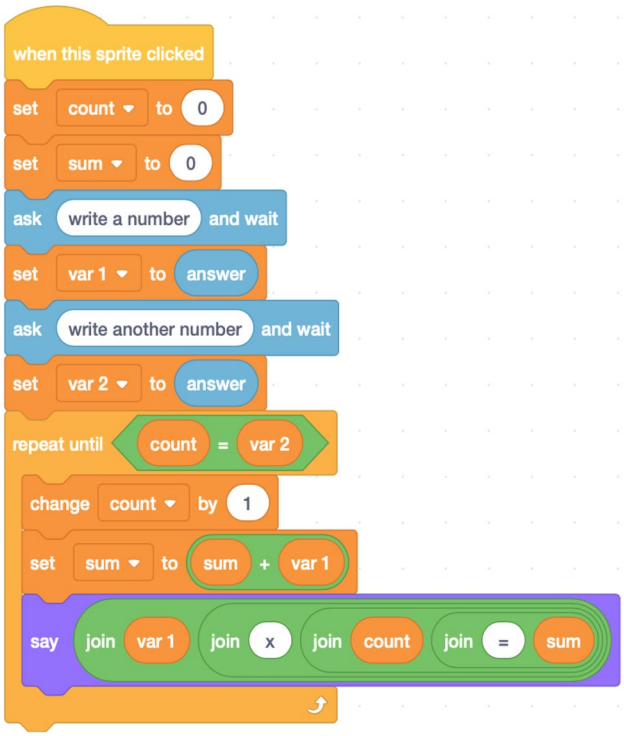

Fig. 6 Scratch code for repeated addition

We do not know what the teachers expected when asking students to compress the code, but we do know through their evaluation of the lesson that no one managed to do this.

In the Scratch code the two variables *count* and *sum* are named and given starting values 0. Then the number to be repeated and the number of repetitions are introduced as the two variables *var1* and *var2*, and given values through the interactive *ask* and *answer* blocks. Finally, a loop is constructed where the count variable is gradually increased until it is equal to *var2*. For each iteration the sprite will write the relevant multiplication on the screen. 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* which change their values during the program. The *count* variable increases in order to keep track of the repeated additions and the *sum* variable accumulates the repeated additions. The final value stored in the variable *sum* is the result of the repeated addition, that is, *sum* is an output variable.

7.2 Core issue: a counter variable

In this example we have many variables, and issues arise concerning the transitions from implicit to explicitly named variables, including the use of a counter variable. There are also a number of issues related to Scratch syntax, as follows: the use of different coloured blocks; what to place inside and outside the loop; and differentiating between input and output.

Whereas a human can keep track of two or even three things simultaneously, a computer needs to be told a sequence of steps. Consequently, names or labels are used for locations in which a value is stored, and which can be accessed by reference to the label. The label stands for the value currently within the location. In programming, naming a variable (for example *var1*) and assigning a value to the variable (here 'answer') are two different processes. Unlike in algebra, the name of a variable often reflects its content and gives the programmer a better understanding of the purpose of the variable, such as the *count* and *sum* variables in this code.

The *answer* block is a temporary variable, although not identified as a Scratch variable in an orange block since it is tightly bound to the sensing block *ask*. Answer is needed as an intermediate step between the value entered by the user and the computer's assignment of this value to the variable *var1*. The next question generates a new answer, and when the value of answer is retrieved for *var2*, it refers to the second answer. In contrast to algebraic notation, this is an example of how variables with the same name can refer to different values in the same code.

The idea of using a variable as a counter in order to keep track of a repeated action is perfectly natural. Young children learn to do it when asked how many objects are made from combining two known piles, starting with the larger number and then counting-on the further number, and subtracting by counting-down from one number to another while keeping track of the number counted. What in programming is a straightforward use of the same idea in a different register, in the algebra register needs a complex collection of signs and subscripts to be expressed effectively, increasing the gap between intuition and algebraic notation:

$$S_1 = a; S_{n+1} = S_n + a = \sum_{1}^{n+1} a$$

It would make sense to rehearse the actions of counting-on and counting-down with learners in order to bring to the surface the idea of a counter which keeps track of the number of repetitions, and the idea of a variable which stores the current value. In terms of early algebra, the important realisation for learners is the use of symbols to stand for as-yet-unknown or as-yet-unspecified quantities, and the possibility of expressing general relationships using these labels.

The task as conceived by the teachers emphasised the what and how of syntax, without much reflection on why the students should work on syntax. In a CT logos it may make sense to delay explicit instruction in syntax, treating it either as a puzzle to be resolved by tinkering with examples,

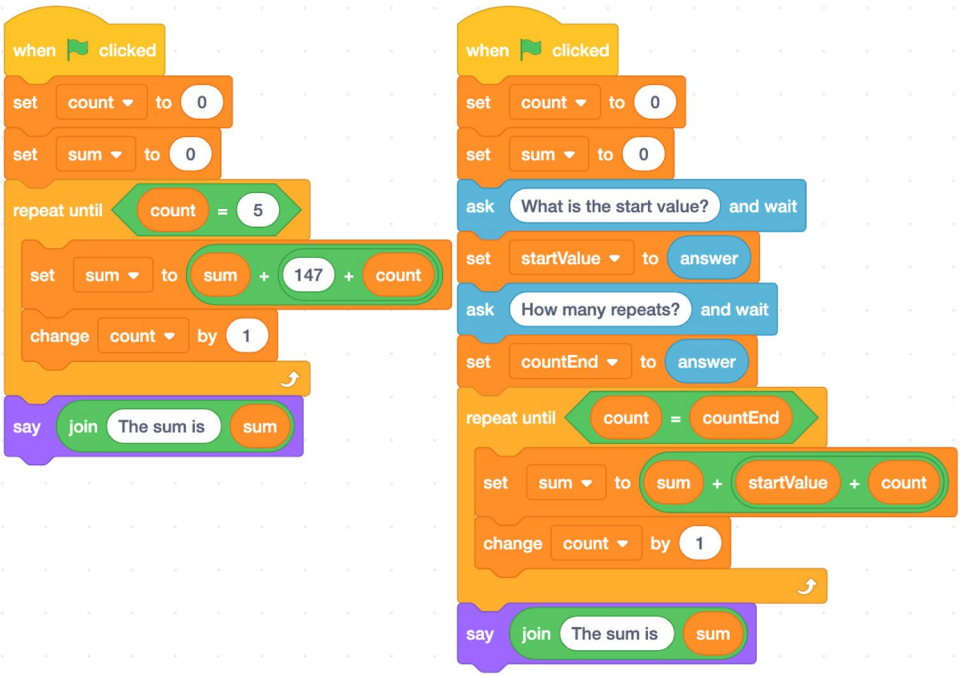

Fig. 7 Scratch codes calculating a specific and an arbitrary sum of consecutive numbers

or as an afterthought for reflection once templates have been adapted by learners.

7.3 Task modifications and possible didactic actions

In this modification we introduced an algebra logos by offering an alternative focus in setting up a counter to repeat a more complex action a specified number of times. Adding up 7 copies of 3 presents neither sufficient challenge, nor sufficient utility to trigger motivation or excitement. It may have been chosen so as not to complicate things, but one advantage of integrating programming with mathematics is to enrich and extend the mathematical scope by making use of fast machines. Therefore, the first modification was to increase the numbers. When the lesson starts, students might find on the board: $147+147+147+\cdots$ going on as long as will fit on the board, and they might be faced with the question 'There are 239 terms here, how shall we get the computer to help us?'. Then, since the repeated addition is no longer needed once the connection to multiplication has been made, a more interesting use of a counter could be to generate the sum of consecutive numbers. For example, $147+148+149+150+\cdots$. While the first is readily spotted as answered using multiplication, which can be done with a calculator or by hand, the second involves much more work.

The didactic choice of task details is to make the task just out of reach of the students' immediate capacity, while remaining within their familiar actions. The teacher can take responses, acknowledging that this looks like a lot of work, and suggest trying a simpler example, in order to see how the programming works. The code to the left in Fig. 7 calculates the value of the five consecutive numbers $147+148+\cdots+151$ by using a loop and a counter variable. With an algebra logos in mind, the teacher can now initiate an analytic discussion about how to turn this particular example of Scratch code into a more general code to calculate any sum of consecutive numbers. This is represented by the code to the right in Fig. 7 where the starting and ending values have been implemented as variables.

Didactically it is vital for learners to articulate the role of each of the statements. One issue is what names to use. In algebra single letters are preferred, whereas in programming words that remind you what role the value has are preferred, such as *startValue* or *countEnd*. This could be a useful topic for discussion and a further translation from words to single letters could be a path into algebraic notation.

A counter variable can be a very useful idea for exploring numbers and experiencing patterns. Similar tasks could include, for example, the following cases: adding three each time; doubling and subtracting one each time; adding the next positive integer or the previous two numbers together. The same idea could then be revisited as students encounter a wider range of numbers. Changing the task so as to present an unexpected phenomenon means that the program makes exploration and example-construction possible, which might provide motivation to engage in programming as well as in expressing generalities, picking up syntax as it is needed.

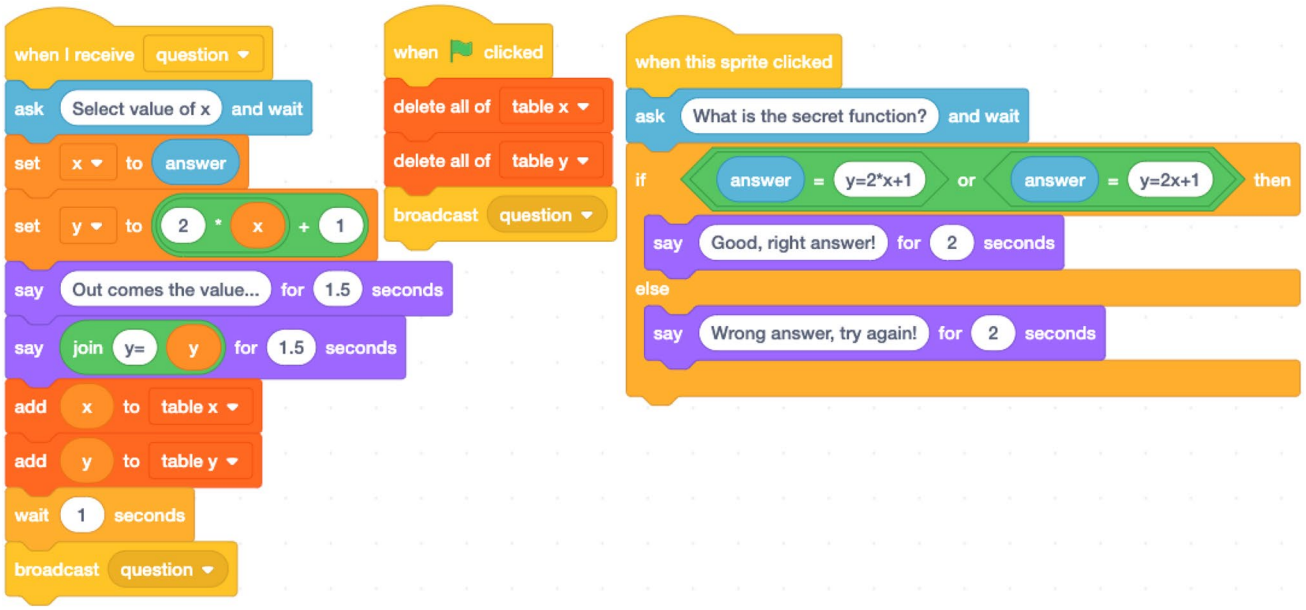

Fig. 8 The original function machine code given to the students

8 A function machine.

The third lesson was designed by two teachers, each trying it in their own grade 6 class (age 12). Rather than being a standalone activity, the lesson was part of a sequence of lessons about functions, spanning several weeks. Before the programming activity, the students had been working with what the teachers referred to as unplugged programming, using a box to represent a function machine, with one student outside the box choosing input values and another inside, calculating output values according to a 'secret rule'. By introducing programming, the teachers wanted the students to deepen their understanding of functions. The students had some, but limited, prior experience of Scratch. The teachers explained the learning objectives in the following way: 'The goal is for all to be able to remix the starting code and create a function that makes the program calculate the y-value for an inserted x-value'. The original code gives a standard linear function, that is $y = kx + m$, where both k and m are whole numbers. For students who need a greater challenge, the teachers thought the task could provide opportunities to create other types of functions where for example $k < 1$ or $m < 0$, which could result in functions such as $y = \frac{x}{2} - 4$.

8.1 Initial task

The codes in Fig. 8 were shown to the students, and together the class discussed the questions: 'Where in the code do we find the function?', 'In what ways can the code be changed?' and 'Why is x (referred to as) a variable?'. They were then asked to alter the code in order to construct a new function. The given code consists of three parts: one part resets the two lists and starts off the code asking for values when the green flag is clicked (centre Fig. 8); one part repeatedly asks for a value of x, calculates the corresponding y-value for $2x + 1$ and stores the values in two lists (left in Fig. 8); one part asks the user what the secret function is and reveals whether the answer is correct (right in Fig. 8).

8.2 Core issue: variables and parameters

Moving the hands-on function machine activity into a programming environment introduces a new register and new syntax, and thus changes the knowing-how of the task. It is still the same task, where one person knows the rule and another person suggests input values and has to guess the rule, except that the computer executes the calculations. Not doing the arithmetic could help students attend to structure, as recommended by Hewitt (2019).

In their evaluation, the teachers noted that 20% of the students only copied the original code and struggled to connect the activity to their previous work on functions, while 74% constructed other linear functions and 6% came up with nonlinear functions such as $y = x^2 + 2$. Letting students alter the code and tinker with it opened up opportunities to choose values not only for x and y, but also for k and m, which for some students changed the knowing-what of the task to include integers or rational numbers, or nonlinear functions. Here the programming environment supplied valuable opportunities for exploration. However, without also incorporating a graphical representation, the exploration is hampered, with small chances to 'guess the rule' of a

nonstandard function simply by looking at the table of values created by the program. The focus of the task therefore shifts from 'finding the rule' to 'creating the rule'.

Linear functions are typically introduced in Swedish schools by the equation $y = kx + m$ representing a straight line where k is the slope of the line and m is the y intersect. The symbol x is called a variable, which in turn is described as a letter that can assume different values. For the students this might create some uncertainty, considering that what they have at hand is an expression with four different symbols (k, m, x, y). In algebra, it is possible to mark the difference between variables and parameters by using the notation $f(x)$. From this we know that x is the variable since we have 'a function of x', and that the rest of the symbols are parameters. A parameter is thus another type of variable that can, but does not need to, be assigned a value.

8.3 Task modifications and possible didactic actions

Small changes of the original task could reframe it within an algebra logos, where the learning objective is to experience generality, and where students can encounter the power of variables and the various roles variables can play. Indeterminate quantities of different kinds could thereby be symbolized differently and discussed analytically. In the original task, the code needs to be modified every time the function is changed. By representing also the parameters k and m as variables, and by using lists and random values, it is possible to let the program generate different functions. That way the same person can run the program and guess the function. After exploring individually or in groups, the classroom discussion could then focus on the different roles of the variables.

The modified task started with the specific function given in the original task (A), which was generalised to an arbitrary linear function by replacing the constants with variables that were assigned randomised values (B). Finally, we added an interactive element where the user can guess the randomised values (C).

The sensing block *ask and wait* is replaced by two lists and a loop (see Fig. 9). The program starts by resetting the variable x and the two lists. This is necessary since variables in Scratch retain their values when restarting the program. The program then calculates the function value $f(x) = 2x + 1$ for $x = 0$ to $x = 4$ by using the loop block *repeat* five times. For more values choose more repeats.

(B) The code is modified to an arbitrary function $f(x) = kx + m$, where k and m are given random values each time the program is running. This is done in the code to the left in Fig. 10 by using the operator block *pick random*, here limited to integers from 1 to 10. Again, the program goes through the loop five times, the difference is that this time the function consists of three variables; k, x and m, all

Fig. 9 A program that calculates the values of $2x + 1$ for $x = 0$ to $x = 4$ and produces a list of values

represented by round orange variable blocks. The function y is also represented as a variable. Even though Scratch does not distinguish between the variables k, m, x and y, there is a significant difference from a mathematical perspective, since x and y change their values multiple times during the program while k and m are fixed outside the loop. Mathematically, k and m would be considered as parameters while x is an independent variable and y a dependent variable.

(C) An interactive element is added to the program where the user is asked to guess, or actually conclude from the lists, the values of k and m. An example is shown to the right in Fig. 10, triggered by the sprite 'Click to guess function' in Fig. 11. The program produces a table of values which can be made longer in an instant. The students can therefore turn their attention to the structure of the function as expressed in the code or to the patterns that emerge in the table of values.

To stimulate reflection, the students could be asked how they can calculate k and m by using the lists or how they would calculate them if they did not set the x-value to 0, and to describe differences and similarities between x, k and m as the program runs.

In the modified task, the students are asked to generalise a specific function by constructing a code for an arbitrary linear function, which provides opportunities to experience how variables can behave differently and play different roles. This promotes an analytic discussion about the meaning of variable, both in relation to the code and in a mathematical perspective. Students might conclude that only x and y change value as the program runs, whereas k and m only changes when the program starts afresh. Hence, x and y

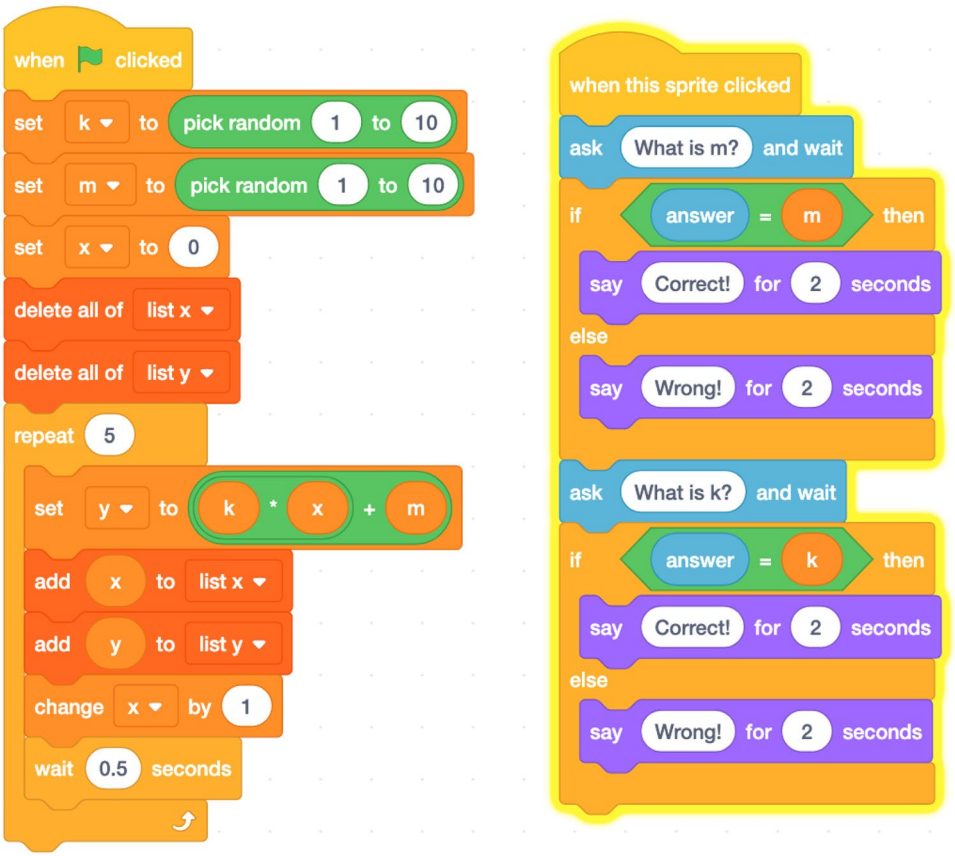

Fig. 10 A program that creates a random linear function (left) including an interactive element (right)

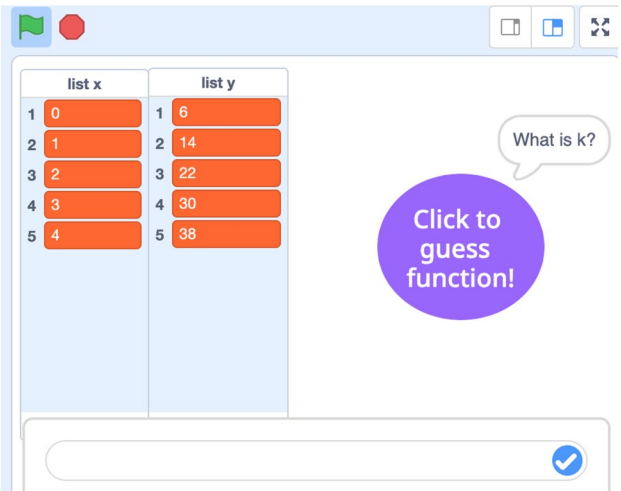

Fig. 11 In Scratch, the two lists generate a table of values

are allowed to vary *within* the program while k and m vary *between* programs.

As stated above, one of the great innovations of symbolic algebra is that it is easy to operate on and manipulate variables as as-yet-unknown or as parameters, without assigning them values. In programming it is not possible to run a code with variables without assigning them values. Therefore, analyticity requires that teachers engage the students in a discussion about the different roles of the variables. Even if the differentiation between variables and parameters may not be expected in early algebra, the task could provide a first intuitive way of thinking about variables that prepare young students for formal algebra later on, which is a key aspect of the early algebra movement (Carraher & Schliemann, 2007).

9 Discussion

By presenting, analysing and tinkering with three examples of authentic school programming activities in the Scratch environment, our aim was to unpack different meanings of variables and shed light on *what might be* in the interplay between programming and mathematics in relation to early algebra.

In Sect. 6 our analysis of the task *Drawing circles* identified implicit variables as a core issue, and found didactical potential in making implicit variables explicit, naming them and exploring effects of letting them vary. We showed that small modifications of the task could bring out such potential. Seeing what modifications are necessary presumes a logos that supplies arguments about what the Scratch code

could make visible, and what aspects of variables could be enhanced through the task. Block-based programming environments like Scratch afford the identification of variables by having specific variable blocks that can be named and explored. Indeterminate numbers are thus symbolised by specific blocks and can be handled as if they were known numbers when creating and tinkering with the code. In the modified task the set numbers are replaced by placeholders, i.e., variables that are possible to change but set for each execution of the program. This conceptual switch from numbers to placeholders is referred to by Ely and Adams (2012, p. 22) as "the birth of symbolic algebra". A constraint of the environment is, however, that when the variables are hidden in other types of blocks they may not be detected and seen as variables, in which case a learning opportunity is lost.

In Sect. 7 we identified the didactic potential of using a *Counter variable* to explore addition of consecutive numbers, and hence of other sequences generated successively, expanding the opportunity of experiencing patterns beyond simple calculations with single-digit numbers. This in turn offers the possibility of experiencing generalisation through changing aspects of the generating function. We argue that this switch of register (in the sense of Duval, 2006) from finger-tallying and pen-and-paper algorithms to the use of a computer code, can serve as an affordance when exploring sequential addition and interesting number sequences exploiting the power of computers. A counter variable can build on children's intuitive ideas of counting-on, but this is a difficult concept to work with using traditional algebraic notation, while a computer code is rather straightforward with its procedural rather than structural approach. In addition, the naming of variables in the computer code can start with meaningful words and gradually lead to single-letter names more often used in algebraic notation. The aspect of naming was found to be an issue already in the early Logo programming experiments in the 1980s (Noss, 1986). Our modifications of the task to name the variables *startValue* and *countEnd* instead of *var1* and *var2* could help early algebra learners to see the role of the variables as generalized numbers.

Section 8 presents a task that builds on the well-known idea of a *function machine*, where a rule (a function) is presented and explored through input and output values. In this task we identified an opportunity to highlight differences between variables and parameters, or what Ely and Adams (2012) refer to as placeholders. In a functional relation like $y = kx + m$ the algebraic notation does not disclose the different meanings of the variables *y, x, k* and *m*. In the Scratch code, their different roles can be seen in where they appear and how they play out as the program is run, thus highlighting differences between parameters and variables. We argue that a register switch, from a metaphorical context in a linguistic and semiotic register to a code in a programming environment, changes the focus and could direct students' attention to specific and different roles of variables. Instead of being only a question of calculating function values and guessing the function, students can begin reflecting on structure and the different roles of the variables. However, to fulfil the conditions for algebraic thinking as described by Radford (2014), the teacher plays an important role in moderating an analytical reflection on these different roles.

Previous research has shown that programming within the context of mathematics in the early grades does not automatically enhance mathematics learning (Benton et al., 2017; Boylan et al., 2018). This was visible also in the lesson study project that provided the data for this paper (Jahnke, 2020; Kilhamn et al., 2021). When analysing the three tasks described here, we concluded that one reason for the lack of mathematical learning opportunities could be that the teachers did not refer to a mathematically oriented logos that justified the choices they made. Explicit focus of attention towards syntax issues or programming concepts and procedures implies a computer-informed logos. In contrast, a mathematics-oriented logos would focus attention on the potential use of programming to enrich mathematical learning. Through our modifications of the tasks we have shown that focussing on powerful algebraic ideas, while letting computational thinking and syntax be subordinate learning objectives, opens up rich opportunities for a programming environment to function as a new register in which to express and explore different aspects of variables in an early algebra context. Since variables do not always have the same meaning and role in algebra (Ely & Adams, 2012; Partanen & Tolvanen, 2019; Usiskin, 1988), or in programming compared to algebra (Bråting & Kilhamn, 2021), exploring and comparing them ought to be made a focus of attention sooner rather than later.

Programming as part of the mathematics curriculum, and computer code as a register to use for mathematical representation, is new to teachers in the lower grades, at least in Sweden. In order to achieve integration of programming and early algebra, our analysis suggests that attention must be paid to the didactic transposition at all institutional levels, from curriculum to classroom (Chevallard, 2006). If the aim is to use programming to learn algebra, knowledge needs to be transposed so that the logos underpinning praxis is explicitly oriented towards algebra. That means building a praxeology where praxis (i.e., tasks and techniques) is allied to a logos that gives arguments supporting the use of programming. We have shown that an algebra logos can enrich some programming activities to potentially enhance algebra learning. We do not know how these would play out in praxis when given to teachers, but we do know that teachers play an essential role in bringing out potential learning opportunities (Sutherland, 1993). Teachers must therefore be

given as much help as possible to see the learning potentials in the activities and digital tools that are provided.

Just as important as how a Scratch code can enrich learning is to know also when it is not useful or even runs the risk of complicating things for algebra learners. Teachers with little knowledge of programming are restrained by a narrow range of available didactic choices when planning a mathematics lesson involving programming, so the praxeology needs to be more clearly developed on the level of the educational system as *knowledge to be taught* (see Fig. 1). When programming tasks are presented to teachers the logos should be as clear as the praxis, and if we want students to develop algebraic thinking, tasks need to be justified by an algebra logos. We call for researchers, curriculum designers and textbook writers to take on this challenge.

Acknowledgements This work was supported by the Swedish Research Council [Grant No. 2018-03865].

Funding Open access funding provided by University of Gothenburg.

References

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.

Blanton, M., Levi, L., Crites, T., Dougherty, B., & Zbiek, R. M. (2011). *Developing essential understanding of algebraic thinking for teaching mathematics in grades 3–5.* Series in Essential understandings, National Council of Teachers of Mathematics.

Blikstein, P. (2018). Pre-college computer science education: A survey of the field. Mountain View, CA: Google LLC. https://goo.gl/gmS1Vm. Accessed 6 Aug 2021.

Bocconi, S., Chioccariello, A., & Earp, J. (2018). *The Nordic approach to introducing computational thinking and programming in compulsory education*. Report prepared for the Nordic@BETT2018 Steering Group.

Bosch, M., & Gascon, J. (2006). Twenty-five years of didactic transposition. *ICMI Bulletin, 58*, 51–63.

Boylan, M., Deack, S., Wolstenholme, C., Reidy, J., & Reaney-Wood, S. (2018). S*cratchMaths evaluation report and executive summary*. Education Endowment Foundation. Sheffield Hallam University. https://shura.shu.ac.uk/23758/1/ScratchMaths%20evaluation%20report.pdf. Accessed 10 Aug 2021.

Brennan, K., & Resnick, M. (2012). New frameworks for studying and assessing the development of computational thinking. In *Proceedings of the 2012 annual meeting of the American Educational Research Association* (pp. 1–25), Vancouver, Canada.

Britt, M. S., & Irwin, K. C. (2011). Algebraic thinking with and without algebraic representation: A pathway for learning. In J. Cai & E. Knuth (Eds.), *Early algebraization* (pp. 137–159). Springer.

Bråting, K., & Kilhamn, C. (2021). Exploring the intersection of algebraic and computational thinking. *Mathematical Thinking and Learning, 23*(2), 170–185.

Bråting. K., Kilhamn, C., & Rolandsson, L. (2021). Integrating programming in Swedish school mathematics: Description of a research project. In Y. Liljekvist, L. Björklund Boistrup, J. Häggström, L. Mattsson, O. Olande, H Palmér (Eds.), *Sustainable mathematics education in a digitalized world. Proceedings of MADIF12* (pp. 101–110), SMDF.

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.

Carraher, D. W., & Schliemann, A. D. (2007). Early algebra and algebraic reasoning. In F. K. Lester (Ed.), *Second handbook of research on mathematics teaching and learning. A project of the National Council of Teachers of Mathematics* (pp. 669–705). Information Age Publishing.

Chevallard, Y. (2006). Steps towards a new epistemology in mathematics education. In M. Bosch (Ed.), *Proceedings of the Fourth Congress of the European Society for Research in Mathematics Education, CERME 4* (pp. 21–30). FUNDEMI IQS-Universitat Ramon Llull.

Davydov, V. (1972/1990). J. Teller, Trans.) *Types of generalization in instruction: Logical and psychological prelims in the structuring of school curricula*. National Council of Teachers of Mathematics.

Duval, R. (2006). A cognitive analysis of problems of comprehension in a learning of mathematics. *Educational Studies in Mathematics, 61*(1–2), 103–131.

Ely, R., & Adams, A. E. (2012). Unknown, placeholder, or variable: What is x? *Mathematics Education Research Journal, 24*(1), 19–38.

Gadanidis, G., Hughes, J. M., Minniti, L., & White, B. J. (2017). Computational thinking, grade 1 students and the binomial theorem. *Digital Experiences in Mathematics Education, 3*(2), 77–96.

Helenius, O., & Misfeldt, M. (2021). Programmeringens väg in i skolan—En jämförelse mellan Danmark och Sverige [How programming makes its way into school—A comparison between Denmark and Sweden]. In K. Bråting, C. Kilhamn, & L. Rolandsson (Eds.), *Programmering i skolmatematiken—möjligheter och utmaningar [Programming in school mathematics—opportunities and challenges].* Studentlitteratur.

Hewitt, D. (2019). "Never carry out any arithmetic": the importance of structure in developing algebraic thinking. 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. 558–565). Freudenthal Group & Freudenthal Institute, Utrecht University and ERME.

Hofmeister, J., Siegmund, J., & Holt, D. V. (2019). Shorter identifier names take longer to comprehend. *Empirical Software Engineering, 24*(1), 417–443.

Hoyles, C., & Noss, R. (2015). *Revisiting programming to enhance mathematics learning. Paper presented at Math + Coding Symposium*. Western University.

Jahnke, A. (Ed). (2020). *Programmering i skolan. Var, när, hur och varför?* [Programming in school. Where, when, how and why?] Slutrapport från FoU-programmet Programmering i ämnesundervisningen [Final report from the FoU program Programming in subject teaching]. Ifous report series 2020:5.

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. (2004). Algebraic thinking in the early grades: What is it? *The Mathematics Teacher, 8*(1), 139–151.

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. ICME 13 Topical Surveys*. Springer Open.

Kilhamn, C., Rolandsson, L., & Bråting, K. (2021). Programmering i svensk skolmatematik [Programming in Swedish school mathematics]. *LUMAT—International Journal on Mathematics, Science and Technology Education, 9*(1), 283–312.

Küchemann, D. (1978). Children's understanding of numerical variables. *Mathematics in School, 7*(4), 23–26.

Mannila, L., Dagiene, 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 Working Group Reports of the 2014 on Innovation & Technology in Computer Science Education Conference* (pp. 1–29). ACM.

Mason, J. (2002). *Researching your own practice: The discipline of noticing*. Routledge Falmer.

Mason, J., Graham, A., & Johnston-Wilder, S. (2005). *Developing thinking in algebra*. SAGE.

Noss, R. (1986). Constructing a conceptual framework for elementary algebra through Logo programming. *Educational Studies in Mathematics, 17*(4), 335–357.

Papert, S. (1980). *Mindstorms: Children, computers, and powerful ideas*. Basic Books.

Partanen, A.-M., & Tolvanen, P. (2019). Developing a frame for analysing different meanings of the concepts of variable mediated by tasks in elementary-school mathematics textbooks. *Nordic Studies in Mathematics Education, 24*(3–4), 59–79.

Pólya, G. (1954). *Induction and analogy in mathematics. Mathematics and plausible reasoning* (Vol. 1). Princeton University Press.

Radford, L. (2014). The progressive development of early embodied algebraic thinking. *Mathematics Education Research Journal, 26*(2), 257–277.

Resnick, M., Maloney, J., Monroy-Hernandez, 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.

Sutherland, R. (1993). Connecting theory and practice: Results from the teaching of Logo. *Educational Studies in Mathematics, 24*(1), 95–113.

Swedish National Agency of Education. (2018). *Curriculum for the compulsory school, preschool class and school-age educare*. Norstedts.

Takahashi, A., & Yoshida, M. (2004). Lesson-study communities. *Teaching Children Mathematics, 10*(9), 436–437.

Treffers, A. (1987). *Three dimensions: A model of goal and theory. Description in mathematics instruction—The Wiskobas Project* (p. 247). Reidel.

Usiskin, Z. (1988). Conceptions of school algebra and uses of variables. In A. Coxford & A. Shuite (Eds.), *The ideas of algebra, K-12. 1988 yearbook* (pp. 8–19). National Council of Teachers of Mathematics.

Watson, A., & Mason, J. (2005). *Mathematics as a constructive activity: Learners generating examples*. Erlbaum.

Wing, J. (2006). Computational thinking. *Communications of the ACM, 49*(3), 33–36.