Received: 7 August 2017 | Revised: 4 December 2017 | Accepted: 22 March 2018

DOI: 10.1111/jcal.12266

ORIGINAL ARTICLE

WILEY Journal of Computer Assisted Learning

Beyond jam sandwiches and cups of tea: An exploration of primary pupils' algorithm-evaluation strategies

L. Benton[1] | I. Kalas[1,2] | P. Saunders[1] | C. Hoyles[1] | R. Noss[1]

[1]UCL Knowledge Lab, UCL Institute of Education, University College London, UK

[2]Department of Informatics Education, Comenius University, Slovakia

Correspondence
Laura Benton, UCL Knowledge Lab, UCL Institute of Education, University College London, 23-29 Emerald Street, London WC1N 3QS, UK.
Email: l.benton@ucl.ac.uk

Funding information
Education Endowment Foundation

Abstract

The long-standing debate into the potential benefit of developing mathematical thinking skills through learning to program has been reignited with the widespread introduction of programming in schools across many countries, including England where it is a statutory requirement for all pupils to be taught programming from 5 years old. Algorithm is introduced early in the English computing curriculum, yet there is limited knowledge of how young pupils view this concept. This paper explores pupils' (aged 10–11) understandings of algorithm following their engagement with 1 year of ScratchMaths, a curriculum designed to develop computational and mathematical thinking skills through learning to program. A total of 181 pupils from 6 schools undertook a set of written tasks to assess their interpretations and evaluations of different algorithms that solve the same problem, with a subset of these pupils subsequently interviewed to probe their understandings in greater depth. We discuss the different approaches identified, the evaluation criteria they used, and the aspects of the concept that pupils found intuitive or challenging, such as simplification and abstraction. The paper ends with some reflections on the implications of the research, concluding with a set of recommendations for pedagogy in developing primary pupils' algorithmic thinking.

KEYWORDS

algorithm, children's programming, computing education, mathematics, Scratch

1 | INTRODUCTION

The potential benefit of developing mathematical thinking skills through learning to program has been the subject of debate for several decades (Du Boulay, 1980; Hoyles & Noss, 1987a; Hoyles & Noss, 1987b; Noss, 1987b). In recent years, this debate has been reignited due to widespread changes in computing/informatics within the school systems of many different countries with an increased emphasis on learning to program from an early age (Bocconi, Chioccariello, Dettori, Ferrari, & Engelhardt, 2016; Gujberova & Kalas, 2013; Kabatova, Kalas, & Tomcsanyiova, 2016; Passey, 2016). Researchers have struggled to agree whether programming benefits mathematical understanding or not, due in part to the crucial role of teachers. However, there is evidence that programming can benefit the learning of specific areas of mathematics such as algebra (Noss, 1986), geometry (Noss, 1987a), ratio and proportion (Clements & Sarama, 1997) as well as more general conceptual and affective issues such as self-confidence and mathematical discussion (Howe & O'Shea, 1978). Clements (1999) has also noted that mathematics learning is most successful in studies that "involve carefully planned sequences of computer programming activities." Clements (1999) suggests that exposing pupils to computer programming is not enough and there is a need for a curriculum explicitly designed to exploit the connections between programming and mathematics.

From September 2014, all primary schools in England have been required to teach the national computing curriculum, which includes designing and building programs. There are challenges in implementation with limited guidance on how to teach the proposed content, the

specific levels of knowledge or understanding pupils should achieve at each stage of the curriculum and issues pupils are likely to encounter and how these should be addressed (Passey, 2016). Further challenges concern how to fit the new curriculum content into an already busy timetable and crucially how to forge cross-curricular links from computing to other curriculum areas.

The ScratchMaths (SM) project aims to address some of these challenges by providing a comprehensive curriculum for Year 5 and 6 pupils (aged 9–11) that maps directly to the computing curriculum, seeks to develop pupils' programming skills as well as exploit these skills to explore key mathematical concepts with explicit links to the mathematics curriculum.

Algorithm underpins the English primary computing curriculum, with pupils expected to "apply" this concept throughout their computing lessons. Algorithm can also be found under another guise within mathematics where parallels can be drawn with procedural and logical reasoning. Here, pupils are expected to follow through a logical argument, which in mathematics is shaped by the representations used to express reasons and by classroom conventions (see, e.g., the discussion of "proofs" that show that the sum of two odd numbers is always even in Healy & Hoyles, 2000). The SM curriculum has thus been built so that algorithm, as instantiated in a computer program, serves as an overarching means to forge connections between the two curricula. Given its centrality, we have researched pupils' understandings of algorithm and specifically probed pupils' strategies for evaluating the differences between algorithms that solve the same problem, and the criteria they privilege in their judgements: What aspects of the concept do they find intuitive or challenging and what are the implications for teaching.

2 | BACKGROUND

2.1 | Defining algorithm within computing and mathematics

Algorithm, both the word and the concept, has a long history and is a foundational concept within computer science (CS). It has been suggested that "the concept of algorithm should be considered to be the first axiom of computer science" (Serafini, 2011) and an ability to think algorithmically is a crucial prerequisite of computer programming (Futschek & Moschitz, 2010). Despite being formalized in the 20th century by mathematicians and computer scientists such as Hilbert, Gödel, Church, Post, Turing, and others, in formal CS, at present, the concept of algorithm is rarely rigorously defined (Moschovakis, 2001).

In CS, functions for which an effective method to calculate their values exist are called algorithms. They must consist of a finite number of exact instructions, terminate after a finite number of steps when applied to an input, and produce a correct answer when instructions are followed correctly. More informal definitions of algorithm are widespread in various computing education contexts, but usually focus on the "exact instructions," "finite computation," and "correct answer" aspects of algorithm, and on the computational constructs needed to describe or represent algorithms. For instance, Misfeldt and Ejsing-Dunn (2015) refer to "systematic descriptions of problem-solving and construction strategies, cause-effect relationships, and events." In contrast, Dwyer, Hill, Carpenter, Harlow, and Franklin (2014) focus on the sequence of steps chosen to solve a problem efficiently. Furthermore, within her much cited definition of computational thinking, Wing (2011) refers to algorithm as "an abstraction of a process that takes inputs, executes a sequence of steps, and produces outputs to satisfy a desired goal." The highly theoretical concept of algorithmic solvability is simplified in primary and secondary education to ensuring students have an awareness that one problem-solving strategy may solve a subset of (seemingly unrelated) problems and that a subset of problems may have no solution.[1]

Ideas for the teaching of algorithm have been proposed, usually based on expert knowledge and experience. For example, Futschek and Moschitz (2011) specify several fundamental concepts that should be addressed during primary school children's initial learning about algorithm: including basic commands, their sequence, alternatives (if), iterations (loop), and abstraction (method[2]). They also suggest several stages of learning in algorithmic thinking: interpret, step through and predict the outcome of a given algorithm (understanding); generate own algorithm to achieve a desired result (design); and adapt an algorithm for solving a specific problem to more general problems (generalization/simplification). We have endeavoured to instantiate these ideas as part of the pedagogical approach in the SM curriculum (see Section 2.2), and through the empirical research reported here, we intend to test some of the assumptions in this approach from the pupil perspective.

From a young age, pupils in England encounter the idea of algorithm, with the Key Stage 1 (aged 5–7 years) computing curriculum expecting pupils to create their own simple algorithms and debug them, as well as to employ logical thinking to step through an algorithm and predict the outcome. Considering the complexities of algorithm highlighted above, it is somewhat surprising that the concept is introduced so early and this raises concerns about the potential for trivialization in the interests of making it accessible to this age group.

As pupils move into Key Stage 2, they are expected to build on the knowledge of algorithms developed earlier, particularly in the design of programs where they are required to understand and use sequence, selection and repetition, influencing the order in which the steps of an algorithm would be run. A further objective related to the concept of algorithm is "Use logical thinking to explain how some simple algorithms work and to detect and correct errors in algorithms and programs."[3] Thus, pupils would be expected to be able to *explain* their own algorithms as well as interpret and predict the result of someone else's algorithm. It is hoped that with an increased focus

[1] In his *On the Calculation with Hindu Numerals* (written about 820 and translated later into Latin as *Algoritmi de numero Indorum*), a Persian mathematician Al-Khwarizmi presented useful problem-solving methods with applications to a wide set of problems: Hence, the concept is named after him. Although we do not pursue this further here, the concept of algorithm—as illustrated earlier—constitutes an important touching point between mathematics and computer science.

[2] Known as *definition* in some programming environments such as Scratch (introduced later).

[3] For full programme of study, see https://www.gov.uk/government/uploads/system/uploads/attachment_data/file/239033/PRIMARY_national_curriculum_-_Computing.pdf.

on the development of logical and algorithmic thinking skills, pupils would move from debugging code through a trial and error approach to following a planned logical process.

Many pupils experience difficulties in the understanding of algorithms (e.g., Tsalapatas, Heidmann, Alimisi, & Houstis, 2012), which is unsurprising considering the variation in simply defining the term as well as the complexity of the concept. However, despite the importance of algorithm in CS, there is limited research into the nature of these specific difficulties, what features help young pupils to interpret algorithms and the criteria they use for evaluating similar algorithms. To understand the meanings that pupils themselves bring to programming, one important dimension is how they evaluate their own work and the work of others as well as the role the teacher can play within this process. These issues form the core of our research.

2.2 | Algorithm within the SM project

Following engagement in the SM curriculum, this paper seeks to go beyond pupils' *definition*, *understanding*, and *implementation* of algorithms to consider how they evaluate similar algorithmic solutions. We aim to uncover the characteristics pupils prioritize, what they find intuitive and/or challenging, and how teaching practices might be adapted to better support the learning of this concept.

2.3 | Project background

The overarching aim of the SM project is to investigate how learning to program can be exploited as a conceptual framework for mathematical reasoning among pupils aged 9–11 years. The project has involved the development of a 2-year intervention, which addresses key aspects of the primary computing and mathematics curriculums in upper Key Stage 2. The intervention comprises six modules (three per year) and was designed by researchers working closely with four "design" schools to iteratively test and refine the curriculum resources. One goal of the curriculum design was to ensure accessibility across a wide cross section of pupils at different attainment levels and to particularly address the needs of those pupils who struggle with conventional mathematics.

This intervention has recently undergone an independent evaluation, funded by the Education Endowment Foundation, in around 50 schools across England, with the results due to be published in summer 2018.[4]

2.4 | The role of Scratch

Scratch[5] is a programming environment freely available online and widely used both in and out of schools. Scratch is a visual blocks-based language that allows children to build scripts (programs) through snapping together different coloured blocks (commands), thus circumventing to a large extent the syntax errors that caused issues in many earlier programming languages for children such as Logo (Resnick et al., 2009). Conceptual challenges are of course still evident, and in the absence of syntax issues, debugging can mainly focus on the algorithmic level (Foerster, 2016).[6]

The overall pedagogic approach in the SM project comprises five unordered constructs—Explore, Explain, Exchange, Envisage, and bridgE (the "5Es"), which have structured the whole classroom approach to the different activities in the SM curriculum (see Benton et al., 2017). The approach was also shaped by constructionism, that is, to seek to foster learning as building knowledge structures by learners actively engaging together in "constructing a public entity" (Papert, 1980). These entities could be constructs such as "beautiful patterns, interactive art, or computer games" (Misfeldt & Ejsing-Dunn, 2015). Papert suggests that the role of the teacher is to make connections between the children's work and powerful mathematical ideas (Papert, 2000). He also proposes the idea of "playing turtle" in which the programmer acts as the programmable object (Papert, 1980; Papert, 1987) and represents the notion that "learning to program can benefit from attempting both to act as the creator of algorithms and as the performer" (Misfeldt & Ejsing-Dunn, 2015).

2.5 | Introducing algorithm within the SM curriculum

In line with our constructionist approach, the algorithms that pupils explore in SM are not trivial, not known in advance, meaningful to consider and compare to alternative strategies of solution, worth reapplying in other (and sometimes unexpected) contexts, and useful to generalize to broader set of tasks. In contrast, typical introductory examples used in English schools tend to be around describing everyday or school activities (e.g., making a jam sandwich or a cup of tea, or steps in multiplying two numbers), which illustrate only limited characteristics of an algorithm: namely (and only partly), the importance of the order of steps, and sometimes the need for precise language.

In SM, the concept of algorithm is equivalent to a set of formal and precise strategies represented in Scratch scripts. It is introduced from the first module through activities designed around building scripts for two different pattern-stamping strategies. The first algorithm creates circular tile patterns by the sprite (a programmable object) moving around the outside of the pattern and repeating the steps *move–turn–stamp*, whereas the second one creates a pattern by the sprite moving from the centre of the pattern and repeating the steps *move–stamp–move backwards–turn* (see Figure 1).

Both strategies are revisited in different contexts: in drawing regular polygons and circular patterns of dots and dashes and then in constructing unexpectedly complex tile patterns. The strength of the algorithms lies in the fact that they can be repeated with different tiles,[7] repeated several times with multiple tiles, or generalized by replacing basic stamping by a user-defined command. They are also both "state-transparent" so can be used in a straightforward way in more complex patterns. The first algorithm is simpler with fewer blocks but the second algorithm with the addition of one step provides a strategy that is easier

[4]The final evaluation report will be made available here: https://educationendowmentfoundation.org.uk/our-work/projects/scratch-programming.

[5]http://scratch.mit.edu.

[6]The absence of error messages in Scratch is an issue in terms of debugging algorithms as the pupils receive no feedback but just know that "something is not working."

[7]Referred to as "costumes" in Scratch, which are different visual representations of the sprite.

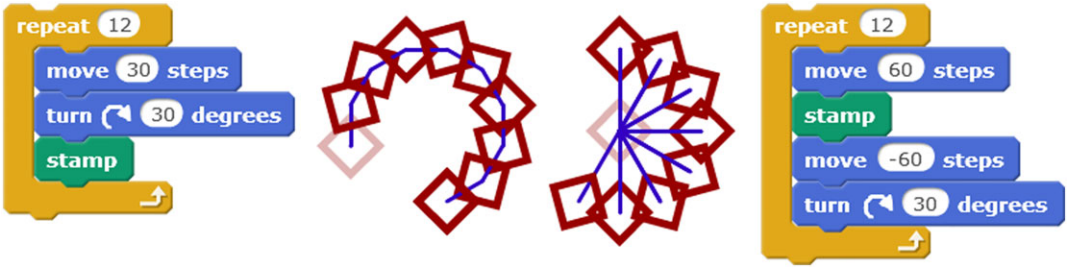


FIGURE 1 Scripts and resulting patterns for the algorithms: 1. move–turn–stamp (left) and 2. move–turn–move back–stamp (right) [Colour figure can be viewed at wileyonlinelibrary.com]

to generalize to other patterns and to extend to build more complex patterns (see Figure 2). In the second algorithm, pupils are also required to use inverse operations connecting to the important mathematical idea of "doing and undoing."

Our goal was to encourage pupils to understand that algorithm was an exact expression of a "strategy." In introductory "unplugged" activities, we use *verbal language* and *body syntonicity* (Watt, 1998) to formulate an algorithm and then move to Scratch itself to formulate it more precisely. For example, pupils are encouraged to stand up and physically enact each step of the algorithm with their bodies as the teacher verbally gives the commands, for example, "*move forward 1 step*," "*turn 45 degrees*," and "*stamp* (*foot*)." Viewing this approach through the lens of constructionism, we are exploiting Papert's idea of an "object to think with," with the Scratch scripts becoming the objects with which to think about algorithms and also employing Papert's idea of "playing turtle"[8]: Encouraging pupils to imagine themselves *as* the sprite walking through each step in order to understand the algorithm. There are clear links here with articulating steps in a logical chain of reasoning required to solve a particular problem in mathematics.

3 | METHODS

We designed a structured paper-based task that was adapted from activities the pupils had already experienced during their SM lessons. The task was intended to tease out algorithmic features that pupils found easy or challenging to interpret, features they gave precedence to as well as their approach to generalizing the algorithms.

All pupils had previously engaged with a year of SM lessons, completing at least the core activities from the first three modules of the curriculum[9] that introduced pupils to algorithm along with computational concepts such as sequencing, repetition, debugging, abstraction, logical reasoning, events, expressions, and parallel behaviours. Pupils explored these concepts through activities that focused on repeating patterns, geometrical drawing, and interactive behaviours. Throughout these three modules, pupils "used" and engaged with mathematical ideas including symmetry, angles, negative numbers, regular polygons, coordinates, multiplication, and factors.

3.1 | Participants

Two researchers visited a diverse subset of the schools (six in total) involved in the SM project to administer the task. A total of 181 Year 6 pupils (aged 10–11) completed the task, with 59 pupils interviewed (mostly in pairs) after the task to explain their answers in more detail. Table 1 provides an overview of the pupils that participated in the task and subsequent interviews. All pupils were taught in mixed-ability classes.

[8]We have brought this back as "playing Beetle" because the programmable object used within parts of the SM curriculum is a Beetle instead of the Logo turtle.

[9]See https://www.ucl.ac.uk/ioe/research/projects/scratchmaths/curriculum-materials for the full SM curriculum that is free to download. In the materials, we identify core activities and extra activities as extensions.

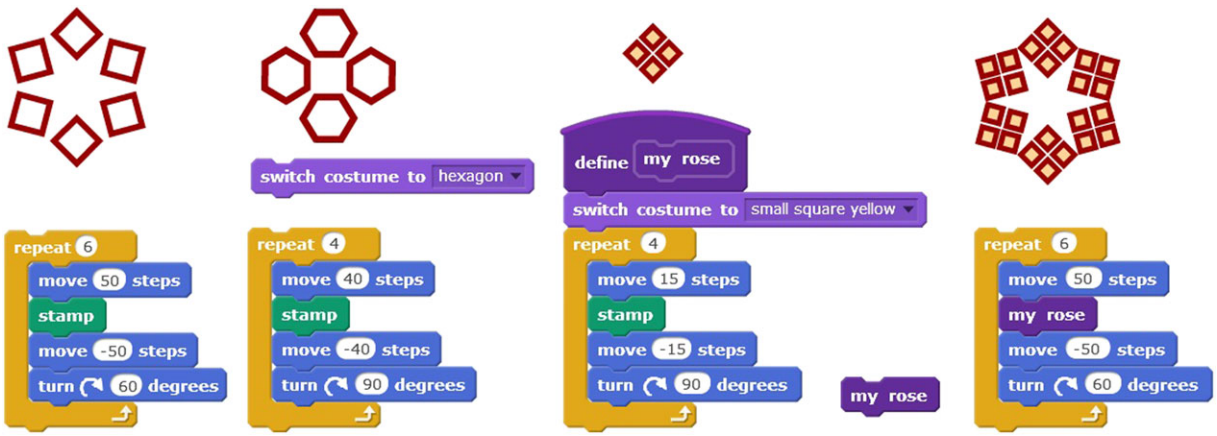


FIGURE 2 The algorithm on the left can be modified by changing the costume of the sprite, or the steps, angle, and number of repeat [Colour figure can be viewed at wileyonlinelibrary.com]

TABLE 1 Overview of schools (EAL = English as an additional language, i.e., non-native English speaker; SEN = special educational needs)

School	No. of pupils completing task	No. of pupils interviewed
School A Large inner-city primary, high EAL, high SEN	27 (16 girls, 11 boys)	3 pairs
School B Large rural primary, low EAL, low SEN	22 (13 girls, 9 boys)	6 pairs
School C Large rural faith primary, low EAL, low SEN	56 (2 classes) (25 girls, 28 boys, 3 unknown)	4 pairs, 1 individual
School D Average-sized urban catholic primary, high EAL, high SEN	21 (10 girls, 11 boys)	5 pairs
School E Average-sized urban faith primary, low EAL, high SEN	26 (11 girls, 15 boys)	4 pairs
School F Large urban junior school, low EAL, low SEN	29 (13 girls, 16 boys)	6 pairs
Total	181 (88 girls, 90 boys, 3 unknown)	57 pupils

3.2 | Design of the task

The pupil task was informed by an earlier version that had been piloted with teachers within the SM professional development sessions. Pupils were given approximately 15 min to complete the task.

Questions 1–3 explored the curriculum objective that pupils should "understand what algorithms are," but we were deliberate in not referencing this in terms of a computer. The questions included

1. What is an algorithm?
2. Give an example of an algorithm.
3. How would you explain what an algorithm is to a younger pupil?

Questions 4–7 explored pupils' evaluations of algorithms (that solved the same problem) in terms of their perceived (a) difficulty, (b) readability, (c) teacher expectation/assessment, and (d) ease of reuse. Five scripts (see Table 2)—each drawing a simple cross (that pupils had previously drawn for themselves in SM)—were chosen for this study as they used different algorithms, had various start and end positions for the Beetle sprite, and a range of control structures and "levels" of abstraction (i.e., incorporated definitions).

Questions 4–6 probed pupils' judgments of these scripts in terms of how "easy" and "easy to read" they were as well as how they thought their teacher would assess the script[10]:

4. Order the scripts from easiest to hardest. Explain your answer.
5. Which script do you find easiest to read? Explain your answer.
6. Which script would your teacher give the best mark to? Explain your answer.

Lastly, Question 7 asked which of the five scripts they would use to draw a fence (as in Table 3) and why, as well as to describe what would the script look like.

[10]A methodology based on that used by Healy and Hoyles (2000) to probe students views of proofs: that is, collect a sample of pupil responses then categorise them.

3.3 | Data analyses

Pupil responses were independently coded by two researchers. For Q1–3 (explaining what an algorithm is and giving an example), the researchers agreed an initial coding scheme:

- a basic definition of an algorithm (using words such as a program, script, set or sequence of instructions, and code);
- an advanced definition of an algorithm (referencing concepts such as multiple strategies to solve the same problem or generalizability);
- categories for the types of examples expected including correct examples in Scratch or "Scratch-like" scripts, other code (e.g., JavaScript), noncode (e.g., a recipe), and incorrect examples (this included answers that consisted of a single command/instruction, i.e., not a sequence).

The interrater reliability was calculated using Cohen's Kappa. First, the coding of the definitions as *basic*, *advanced*, or *incorrect* resulted in an interrater reliability of $\kappa = 0.98$ (very good agreement). Second, the coding of the algorithm examples as *Scratch script*, *other code*, *noncode*, *or incorrect* resulted in an interrater reliability of $\kappa = 0.82$ (very good agreement). Any discrepancies were then discussed and subsequently resolved by the coders.

For Q4–7, the interview responses were transcribed and transferred to nVivo. An initial coding in nVivo was undertaken by one of the researchers first by identifying the script referred to (i.e., A–E) and second by establishing the justification of the classification (i.e., why the script was *Easy*, *Difficult*, *Easy to Read*, would get the *Best Mark*, or could be used to draw the *Fence*). Next, these responses were grouped into high-level themes, which were created as nodes in nVivo and the interview responses were coded according to these high-level themes, which included number of scripts; choice of blocks; number of blocks; use of definitions; input numbers; position of sprite; implementation approach; and demonstrates skills or knowledge. The themes were then further divided into subthemes, which were used to organize pupil justifications for their choice of script in relation to each question.

TABLE 2 Five algorithm scripts included in the task

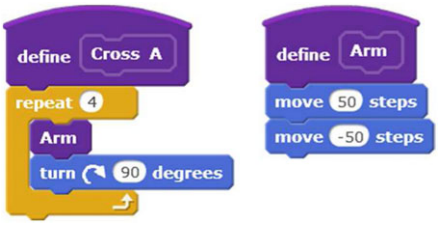
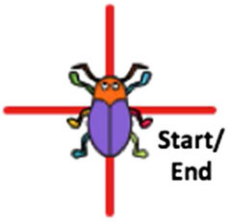
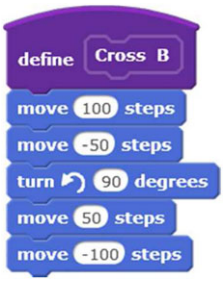
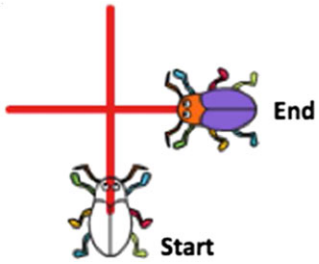
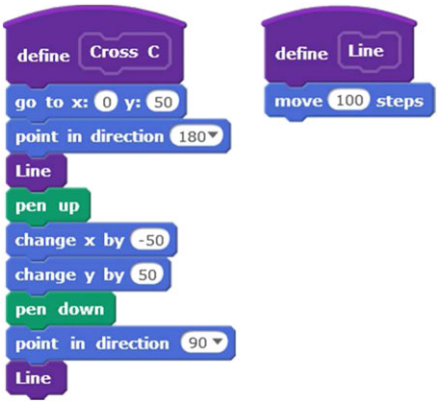
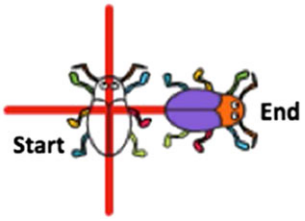
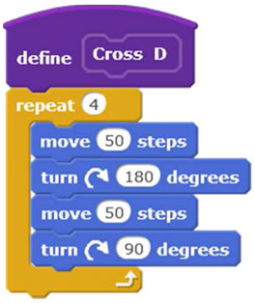
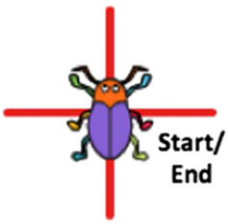
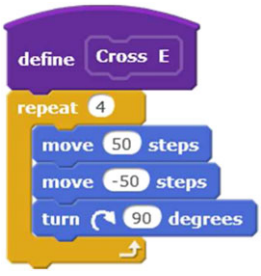
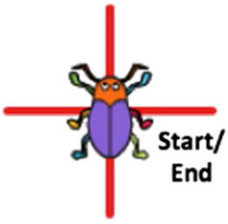
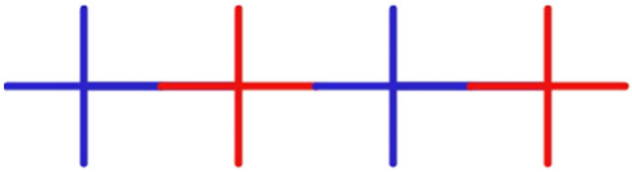
Script	Outcome	Key features
A 		● Same start and end position—drawn from the centre ● Repeat block used in a meaningful way ● Uses a definition to abstract the "arm" of the cross
B 		● Different start and end point ● All steps visible—no repeat ● Single script ● Sprite covers least distance (note—cannot compare with moving via coordinates)
C 		● Different start and end point ● Absolute positioning of sprite ● Uses definition to replace a single block (redundant) ● Pen up ● Greatest variety of blocks ● Models paper and pencil drawing
D 		● Same start and end point—drawn from centre ● Repeat block used in a meaningful way ● Sprite only moves forward (no negative numbers) ● Single script
E 		● Same start and end point—drawn from centre ● Repeat block used in a meaningful way ● Follows same algorithm as Script A but in a single script (no definition)

TABLE 3 Fence created using a repeated row of crosses

Fence script	Key features
	● More complex fence pattern intended to trigger the idea of a repeating component ● Alternating colours to highlight the embedded component ● Engages the consideration of a "construction" plan: where is the start position, where does it end, how do neighbouring components connect ● Highlights power of abstraction by thinking about a component as a one-step subtask ● A challenging question away from a computer

4 | RESULTS

4.1 | How pupils defined algorithm

We begin by examining pupils' understandings of the term "algorithm" (Q1–3). Although all teachers had introduced the term in their SM teaching (during Module 1), many had not subsequently referenced or returned to it within later lessons. This potentially explains why many pupils could not remember or did not know what an algorithm was, as well as the low numbers of pupils attempting these questions (46/181).

Table 4 shows the number of pupils that attempted to answer this question and that were able to provide a basic or advanced definition. Many pupils described algorithm as a "*set of instructions*," but pupils also referred to the idea of it being a "*step by step*" sequence, represented by a "*code*" or "*script*" and used to tell a computer what to do. A few pupils went further indicating the possibility of having multiple solutions ("*when you can do multiple different ways to solve it*") and generalizing to different contexts ("*starting with any type of code and experience the use of different types of blocks and connect them*").

Table 4 also provides an overview of the different types of examples pupils chose as an illustration of an algorithm. The majority of pupils used Scratch scripts as examples, but some pupils used examples from Minecraft, HTML, and Code Studio as well as more general instructions such as "*go straight*, *turn left*, *go up*." Most incorrect examples were where pupils only gave a single command as the example, but there were also examples in the form of a pattern and a description of the instructional language used in an algorithm "(*modal verbs*) [*e.g.*] *must*, *put*, *now*, *should*, *will* (*a command*)."

4.2 | Comparing similar algorithms

In the second part of the task (Q4–6), pupils were asked to evaluate the Scratch scripts against various constructs. In this case, Scratch is the language for expressing the algorithms and the pupils are asked to comment on the different strategies employed to achieve the same solution.

Table 5 shows that similar majorities of pupils selected either Script A or Script E as the "easiest" (from Table 2). The key difference between these scripts was the inclusion of the *Arm* definition, generalizing the move forward and move backwards steps into a new block (although the steps that the sprite follows are the same). However, there were some differences between schools with over 60% of pupils in Schools C and D choosing Script E and 59% of pupils in School B and 80% in School A choosing Script A.

In all schools, a clear majority of pupils (70%+) selected Script C as the hardest. There was a similar split between Scripts A and E in terms of ease of reading, with the majority of pupils in each school giving the same script as both the easiest and easiest to read, although overall, the number saying Script E increased for ease of reading. The majority of pupils in four schools (Schools C–F) thought that their teacher would give the best mark to Script C; however, in Schools A and B, more pupils (50% and 45%) thought their teacher would award the best mark to Script A.

Lastly, in Q7, when asked which script would be most effective in helping them draw a row of four crosses (a fence), the majority of pupils thought that Script C would be most helpful. Below, we describe pupils' reasoning for these choices.

4.3 | What makes an algorithm easy or difficult?

The majority of pupils who were interviewed selected either Script A or E as the easiest script (which was representative of the wider results), with pupils generally saying that the fact these scripts had the smallest number of blocks made them easier (see Table 6). Pupils explained that having fewer blocks made it easier to build, to understand, and to modify as well as more efficient.

TABLE 4 Overview of pupils understanding of the term "algorithm" (G = girl; B = boy)

School	Proportion of pupils attempted question	Basic definition	Advanced definition	Appropriate examples
School A	3/25	3/3 (1G, 2B)	0/3	Scratch (3)
School B	4/22	3/4 (2G, 1B)	0/4	Scratch (3), Incorrect (1)
School C	12/56	11/12 (1G, 9B, 1 unknown)	1/12 (1B)	Scratch (5), Other (1), Incorrect (2), None (4)
School D	2/21	1/2 (1G)	0/2	Other (1), Noncode (1)
School E	1/26	1/1 (1B)	0/1	Other (1)
School F	24/29	20/24 (9G, 11B)	1/24 (1B)	Scratch (14), Other (1), Noncode (2), Incorrect (5), None (2)

TABLE 5 Proportion of pupils selecting each script

Question	Script A	Script B	Script C	Script D	Script E
(4) Easiest	75/179	9/179	4/179	21/179	70/179
Hardest	6/179	10/179	156/179	5/179	2/179
(5) Easy to read	68/176	12/176	3/176	11/176	82/176
(6) Get best mark	40/167	13/167	84/167	9/167	21/167
(7) Use to draw fence	17/129	22/129	60/129	10/129	20/129

Similarly, Table 6 also highlights that having a large number of blocks was the reason that the majority of pupils chose Script C as the most difficult. Pupils also talked about the choice of blocks within the script as being a factor including having a higher diversity of blocks that could add to the difficulty of building the script in addition to understanding.

Furthermore, pupils found the choice of the blocks increased the difficulty because they were complicated or unfamiliar, which included the *point in direction* and *pen up/pen down* blocks as well as the *go to x … y …* block, with a few pupils finding this complicated because it used coordinates.

Of those pupils who selected Script A as the easiest, many stated that it was because of the use of a definition. They explained this helped to shorten the script as well as made it easier to read and quicker to build. A few pupils also referred to the reusability of the defined block increasing the simplicity. Responses to the use of definitions were however mixed and did not make it universally easier for all pupils, with some explaining they found Script E easier because it did *not* have a defined block.

Pupils also raised the redundancy of some blocks as adding to the perceived difficulty, particularly in relation to the use of an unnecessary defined block, which replaced a single block in Script C.

A few pupils mentioned the inclusion of specific functionality in making scripts easier, particularly the use of repeat, which helped in reducing the number of blocks within a script. A small number of pupils also mentioned that the inclusion of familiar blocks, the sprite starting and ending in the same position, and the choice of turn angles all contributed to the ease of understanding an algorithm.

4.4 | Reading algorithms

Pupils again chose Scripts A or E as easier to read, stating that shorter scripts made reading a script easier (see Table 7).

Similarly, pupils were split on the use of definitions to support readability. For some pupils, having everything in a single script made it much easier for them to read. However, others preferred to use definitions, with a few pupils stating that even though they thought Script E was easier overall, including the definition actually made Script A easier for them to read.

A few pupils discussed the choice of blocks impacting on the readability of a script. They found it easier to read blocks with which they were familiar and had used before. They also mentioned specific blocks such as repeat helping.

TABLE 6 Key subthemes for perceived ease or difficulty of an algorithm

Subtheme	Justification	Example quote
Easy		
Small number of blocks	Easy to build, understand, and modify	"*I thought [Script E] was the easiest because it looks simple, it doesn't have like that many steps to it, it just has about around four to five blocks.*"
	More efficient	
Use of definition	Shortened script	"*The arm is defined so you don't have to put that piece of script in. And all you need to do is just turn 90 degrees right and just do arm and just repeat it.*"
	Made it easier to build	
	Reusable	"*Because maybe if you want to do a different script you could use the same block.*"
No definition	Single script	"*Because I thought it was a little bit easier to follow—just all in one block—instead of it having in two different places, so you have to follow in two.*"
	Quicker and easier to build	
Use of repeat	Reduces number of blocks	"*And also the repeat block helps it to repeat and you don't have to write all the steps again and again.*"
Difficult		
Large number of blocks	Harder to understand	"*I thought because [Script C] was quite long and it would make Scratch a bit boring. If you could use a quicker way it would, I mean you're always thinking of quicker ways to do things and you're thinking of the best ways to do things and how it's going to be fully complete and that's why I think the shortest codes can get more out of Scratch.*"
	Takes longer to build	
High diversity of blocks	More difficult to build	"*Because you have to go into loads of things and get a load of things out and then put them all together and it takes … and you can just use repeat to make it shorter.*"
Choice of blocks	Increased difficulty if they were complicated or unfamiliar	"*Because it was so much more complicated to follow, with x and y; to visualise it was harder.*"
Redundancy of blocks	Unnecessary define block	"*It unnecessarily creates the line, which line for a single block, so that's just like making a value for something that was only a single block, which defeats the point of creating a block and then it uses lots of unnecessary codes that it doesn't really need.*"
	Adds complexity	

TABLE 7 Key subthemes for readability, teacher assessment, and extension of algorithm

Subtheme	Justification	Example quote
Readability		
Shorter scripts	Easier to remember	"*Because you can remember that because it is a small script but when it is a big script you forget what the first one is.*"
No definitions	Less to remember	"*Yeah, you have to find what arm means and then so you have to keep that in your head and figure out what that is at the same time.*"
Use of definitions	Made it shorter	"*Because when I saw the sheet I saw that it told me where define arm was so I was thinking well that's pretty easy to read because if that wasn't there you wouldn't know well, what's in the arm but that's there so it's easier to read for me.*"
Choice of blocks	Familiarity	"*It's because mostly all of the ones [blocks] in E we've already like looked [teacher] showed it to us.*"
Use of repeat	Less to read	"*I like it with the repeat blocks, so like you don't have to read it. Like if it was ten, you wouldn't have to read it ten times.*"
Teacher assessment		
Longer scripts	Demonstrated effort	"*Because [Script C] is the longest and you've got like more like work and like if it works it's really good and you know how to do like hard stuff.*"
Shorter scripts	Simpler	"*[Script A] might be like short but might make the sprites do something really awesome, like she might think it's impressive, so short but powerful.*"
Use of definitions	Simpler Shorter script	"*I think she would like A the best because it is quite simple, and she would probably prefer if you use the define, it shortens the sequence.*"
Choice of blocks	Demonstrates complexity, advanced understanding, and/or creativity	"*Because then you're using more blocks and she can tell that you've obviously understood more about it and things like that.*"
Efficiency	Simplest, quickest way	"*I put E again because in coding and maths you're always looking for the simplest, quickest way to do things, E has that, it's very efficient and fast and it gets around the problem quickly.*"
Extension		
Finish position of sprite	Finishes in the nearest place to start drawing the next cross	"*I put C because like, you know on the picture it shows like you go, then it ends there so you might have to like tweak it a little bit but you could just make it carry on drawing another.*"
Same start/finish position	Need to add the least number of blocks	"*I would use A, D or E because unlike the others that start in the same position so you don't need to use code to get them back into the same position and you could simply then use A go t, then you could simply use change x by negative 50, negative 100 I mean, to get it to the next place and it would come back into the same place and you'd keep getting it along and along and along.*"

4.5 | Teacher assessment

Table 7 shows that Script C was the most likely to be chosen as receiving the best mark or seen as a greater achievement by the pupils' teacher. However, a high proportion also selected Script A to receive the best mark. The reason for this difference was reflected in the split opinion between pupils about whether their teacher would prefer a longer script, demonstrating effort, or a shorter script, demonstrating they had considered simplification. In relation to this, some pupils talked about their teacher encouraging them to use definitions in their scripts to make them simpler and giving them credit for this.

Pupils also talked about the choice of blocks within Script C in terms of the script complexity as well as the creativity and advanced understanding demonstrated in generating an alternative solution, which would be rewarded with a good mark. One pupil discussed the efficiency of the script, believing that Script E was the most efficient and therefore would receive the best mark.

4.6 | Extending algorithms

In the final question, pupils were asked to consider which script they would use to draw a fence (of crosses), intended to probe their understanding of extending an algorithm to use in other contexts.

Table 7 highlights that many pupils found this question challenging and did not consider the additional blocks they may need to add to create the fence. Some pupils could not answer or seemed to select a script randomly (as during the interviews, they were unable to provide a reason for their choice). It was also difficult for pupils to document, with some trying to draw out all of the blocks that the script would contain (see Figure 3).

However, of those pupils that were able to clearly justify their choice, the majority selected Script C because of the finishing position of the sprite (seen in the picture—Table 2), which they explained made it easier to continue with drawing the next part of the fence.

There were a few pupils from four of the schools who considered this in a different way and would select one of three different scripts because they all started and finished in the same position, requiring the least number of blocks to be added. However, they had not been able to correctly specify the complete script for the fence (this may be in part explained by the time constraints of the task).

This question requires an understanding of state transparency, which is implicitly touched upon within SM through activities that involve drawing rows of different shapes, but it would be at the teacher's discretion as to whether they had made this explicit within their teaching practice.

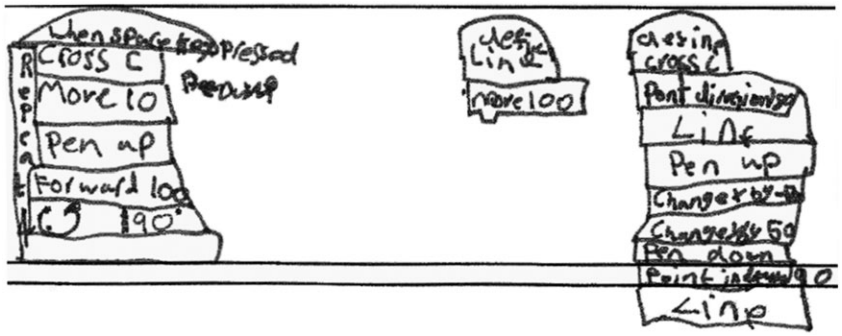


FIGURE 3 Example of a pupil answer to Question 7 (drawing a fence)

5 | DISCUSSION AND CONCLUDING REMARKS

First, we present an overview of pupil understandings of the term "algorithm." Although this is an explicit requirement of the national curriculum in England and is directly referenced in earlier SM activities, it is clear that the use of the term has not been operationalized by several of the teachers. Therefore, although many pupils may be able to apply the concept of algorithm, they struggle to explain what it is at any higher level of abstraction. Knowing the name of something may or may not, of itself, lead to enhanced application of its meaning and role in the broader picture.

It is hardly surprising that the pupils struggled to make sense of the concept of algorithm, and introducing the term so early in the curriculum has the potential risk that pupils would form a view—not entirely unknown in mathematics!—that remembering how to define the word algorithm may be an end in itself. What does naming the making of a cup of tea add to the process unless there is a rationale for so doing? Having a word to express a powerful idea can be the key to unlock the idea in ways that are intellectually empowering if it can be used to build other concepts. This is what we hoped would happen in mathematics—when scripts were used to build mathematical ideas with the reasoning captured in the algorithm.

In addition, our findings have highlighted differences between the criteria pupils are using to evaluate the algorithms against the various constructs; in some instances, there is also a clear difference between classes. Although all of the teachers were following the SM curriculum, generally teaching the same activities in the same order, our findings suggest that some teachers much more explicitly encouraged pupils to employ specific types of strategy that may reasonably be expected to provide some rationale for learning what an algorithm is, how it may be used, and what it "buys" intellectually. An example of this is the practice employed by some teachers who consistently encouraged the use of definitions to simplify scripts, that is, abstraction. In order even to begin to understand the purpose behind this, pupils need to see the sequence of commands (the "body" of the algorithm) as an entity; comparing algorithms necessitates seeing algorithm-as-object so that pupils can say things such as "this algorithm is simpler than that." This would, in theory at least, allow pupils to make quite high-level remarks (implicitly, of course) that amount to statements such as "if (no.blocks-A > no.blocks-B) then B is simpler than A."

In the interview responses, the focus by some pupils on the specific properties of the scripts that included the level of familiarity (with the individual blocks), (sprite) position, length, and diversity (of blocks) suggested they were viewing the script as an "object" in their process of evaluating the algorithms. Pupils experienced challenges in relation to certain representations of sprite position within the scripts, for example, through the direction of heading or through the position on the coordinates grid: Clearly, the "unplugged" experiences were probably a key determinant of pupils' attainment in this regard.

Within the SM curriculum, a subset of activities focuses on the use of definitions within scripts and there is opportunity to utilize them throughout. For some classes, the use of definitions have become a common practice to simplify scripts, reducing complexity such as nested repeats (for more information about the use of definitions within the SM curriculum, see Kalas & Benton, 2018). Our findings suggest that the use of definitions can potentially become an intuitive practice for pupils, but it requires initial facilitation and consistent encouragement from teachers to maintain this practice and to allow pupils to exploit the power of definitions within their algorithms. After such a process, bridging to mathematical reasoning when instantiating processes as objects would be a simple step. For example, a fundamental building block of proving is to be able to reference early findings as objects of the proof.

An interesting finding concerned sometimes conflicting views of the need to reduce redundancy and complexity versus the level of "perceived effort" that had gone into the construction of the different algorithms. This time, we look at the length-is-better criteria, with some pupils maintaining that longer script length and diversity of blocks represented more work and that—irrespective of the readability of the completed scripts—teachers would be thought of as appreciating longer scripts. Others recognized that there was a skill in being able to simplify such a script and removing blocks did not reduce others' perception of the effort/ability reflected in the output. We maintain that this implies a need for teaching explicit focus on the length versus elegance dichotomy to discuss in class the sort of sample scripts and the arguments presented here.

Many pupils struggled with generalizing the algorithms within the last question (Q7) related to drawing the fence. This task highlights a potential challenge in selecting from the pre-existing algorithms the most appropriate strategy for a generalized situation. This is an extension on the more typical activity to specify a generalized algorithm for yourself but is equally important as more advanced algorithmic thinking requires building on the work of others. We know from the extensive work on Logo programming (Noss & Hoyles, 1996) that reuse of code (typically in the form of subprocedures) is not straightforward and takes time to become a normal part of a problem-solving repertoire: for example, young children who are introduced to the idea

of how to construct a SQUARE procedure are reluctant to reuse the code to draw a line of squares or a tower of squares, preferring instead to return to the single "line" strategy that essentially consists of a direct-drive solution surrounded by a definition. Explicitly engaging in discussion about the power of abstraction seems to be an important pedagogy in computing but also in bridging to mathematics.

In light of these findings below, we set out a number of recommendations for primary teaching pedagogy when introducing and extending the concept of algorithm:

- Teachers—and through them the pupils—should understand algorithm as a strategy to solve a problem, or even better—a set of problems. The concept of algorithm should be addressed in contexts (situations) where there may be two or more different strategies to apply.
- The use of simplistic definitions of "algorithm" should be avoided, with pupils allowed to experience the key ideas for themselves before it is labelled.
- The concept of algorithm as a strategy, a way to solve or to proceed should be promoted.
- Pupils should be encouraged in their understanding of "algorithm as object" through unplugged activities.
- Opportunities and explicit strategies for pupils to compare and evaluate similar algorithms should be provided.
- Pupils should be discouraged from thinking longer algorithms demonstrate superior solutions or greater effort and instead encouraged to focus on elegant algorithms that are readable and easy to apply and reapply in different situations.
- Strategies for simplifying algorithms (i.e., abstraction) should be provided.
- Pupils should be helped to understand the power of abstraction through the generalization of algorithms (for instance, through the use of definitions within Scratch).

To sum up, the results of this work show that it is feasible to design activities that scaffold how particular algorithms might be generalized for reuse within other contexts. In so doing, pupils are connecting with an overarching powerful idea, that of abstraction. It is this idea more than any other that confers intellectual power—the encapsulation of code in a definition being perhaps the most basic example of abstraction we have. Our findings highlight how difficult it is for children to compare different approaches and how tackling this problem "from above"—that is, as abstraction in the making—is not being universally translated into the pedagogy of many teachers and something difficult to achieve through using more simplistic metaphors of algorithm. Yet despite these pedagogical challenges, many pupils were able to give some significant and insightful answers that suggest they had started thinking about different strategies, reflecting an early understanding of the concept of algorithm, with pupils not needing any more exact or more formal understanding and/or definition.

It is this knowledge that we believe might provide leverage for the learning of subjects other than computing. If, as is now the case in England it is mandatory for children as young as 7 years old to "understand what algorithms are" and "how they are implemented" as well as appreciating that "programs execute by following precise and unambiguous instructions," it would be surprising if there were no scope to rebuild a mathematics curriculum that exploited this "new" knowledge. In SM, for example, we are attempting to construct learning sequences that *use* knowledge of algorithms to construct mathematical meaning for concepts such as place value, variable, symmetry, and coordinates. Although it is too early to report on the success of this venture, we are reasonably confident that we will at least emerge with an existence theorem that indicates future possibilities for the learning of mathematics and, perhaps, other curriculum subjects.

ACKNOWLEDGEMENTS

We thank the Education Endowment Foundation for funding this work. We also thank our SM colleagues Dave Pratt and Johanna Carvajal for their invaluable contributions to the intervention design. Finally, we are extremely grateful to the teachers and pupils at all of the SM project schools for their continued engagement, hard work, and enthusiasm in trialing our intervention and for sharing some of their experiences with us.

ORCID

L. Benton http://orcid.org/0000-0002-7225-1779

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.

Bocconi, S., Chioccariello, A., Dettori, G., Ferrari, A., & Engelhardt, K. (2016). Developing computational thinking in compulsory education.

Clements, D. H. (1999). The future of educational computing research: The case of computer programming. *Information Technology in Childhood Education Annual*, *1*, 147–179.

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

Du Boulay, J. B. H. (1980). Teaching teachers mathematics through programming. *International Journal of Mathematical Educational In Science and Technology*, *11*, 347–360.

Dwyer, H., Hill, C., Carpenter, S., Harlow, D., & Franklin, D. (2014). Identifying elementary students' pre-instructional ability to develop algorithms and step-by-step instructions. *Proceedings of the 45th ACM technical symposium on Computer science education*.

Foerster, K.-T. (2016). Integrating programming into the mathematics curriculum: Combining Scratch and geometry in grades 6 and 7. Paper presented at the Proceedings of the 17th Annual Conference on Information Technology Education.

Futschek, G., & Moschitz, J. (2010). Developing algorithmic thinking by inventing and playing algorithms. Paper presented at the Proceedings of the 2010 Constructionist Approaches to Creative Learning, Thinking and Education: Lessons for the 21st Century (Constructionism 2010).

Futschek, G., & Moschitz, J. (2011). Learning algorithmic thinking with tangible objects eases transition to computer programming. *International Conference on Informatics in Schools: Situation, Evolution, and Perspectives*.

Gujberova, M., & Kalas, I. (2013). Designing productive gradations of tasks in primary programming education. Paper presented at the Proceedings of the 8th Workshop in Primary and Secondary Computing Education.

Healy, L., & Hoyles, C. (2000). A study of proof conceptions in algebra. *Journal for Research in Mathematics Education*, *31*, 396–428.

Howe, J., & O'Shea, T. (1978). Computational metaphors for children. *Human and Artificial Intelligence*.

Hoyles, C., & Noss, R. (1987a). Children working in a structured Logo environment: From doing to understanding. *Recherches en Didactiques des Mathematiques*, *8*(12), 131–174.

Hoyles, C., & Noss, R. (1987b). Synthesizing mathematical conceptions and their formalization through the construction of a Logo-based school mathematics curriculum. *International Journal of Mathematical Education in Science and Technology*, *18*(4), 581–595.

Kabatova, M., Kalas, I., & Tomcsanyiova, M. (2016). Programming in Slovak Primary Schools. *Olympiads in Informatics*, *10*, 125–159.

Kalas, I. L. & Benton, L. (2018). Defining procedures in early computing education. In: Tomorrow's Learning: Involving Everyone—Learning with and about technologies and computing. A. Tatnall, & M. E. Webb (Eds.), *IFIP AICT*. Heidelberg, Germany: Springer.

Misfeldt, M., & Ejsing-Dunn, S. (2015). Learning mathematics through programming: An instrumental approach to potentials and pitfalls. *CERME 9th Congress of the European Society for Research in Mathematics Education*.

Moschovakis, Y. N. (2001). What is an algorithm. Mathematics unlimited–2001 and beyond, *2*, 919–936.

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

Noss, R. (1987a). Children's learning of geometrical concepts through Logo. *Journal for Research in Mathematics Education*, *18*, 343–362.

Noss, R. (1987b). How do children do mathematics with LOGO? *Journal of Computer Assisted Learning*, *3*, 2–12.

Noss, R., & Hoyles, C. (1996). Windows on mathematical meanings: Learning cultures and computers.

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

Papert, S. (1987). Computer criticism vs. technocentric thinking. *Educational Researcher*, *17*(1), 22–30.

Papert, S. (2000). What's the big idea? Toward a pedagogy of idea power. *IBM Systems Journal*, *39*, 720–729.

Passey, D. (2016). Computer science (CS) in the compulsory education curriculum: Implications for future research. Education and Information Technologies, 1–23.

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

Serafini, G. (2011). Teaching programming at primary schools: Visions, experiences, and long-term research prospects. Paper presented at the International Conference on Informatics in Schools: Situation, Evolution, and Perspectives.

Tsalapatas, H., Heidmann, O., Alimisi, R., & Houstis, E. (2012). Game-based programming towards developing algorithmic thinking skills in primary education. *Scientific Bulletin of the "Petru Maior" University of Targu Mures*, *9*, 56.

Watt, S. (1998). Syntonicity and the psychology of programming. *Proceedings of 10th Annual Meeting of the Psychology of Programming Interest Group*.

Wing, J. M. (2011). Research notebook: Computational thinking—What and Why? The link. Retrieved from https://www.cs.cmu.edu/link/research-notebook-computational-thinking-what-and-why

How to cite this article: Benton L, Kalas I, Saunders P, Hoyles C, Noss R. Beyond jam sandwiches and cups of tea: An exploration of primary pupils' algorithm-evaluation strategies. *J Comput Assist Learn*. 2018;1–12. https://doi.org/10.1111/jcal.12266