An Evaluation of Primary School Children coding using a text-based language (Java).

Abstract

All primary school children in England are required to write computer programs and learn about computational thinking. There are moves in other countries to this effect e.g. the US K-12 Computer Science framework for development. Debates on how to program and what constitutes computational thinking are on-going. Here we report on a study of programming by children aged 7 – 11 using Java, and elements of computational thinking they experience. Our platform comprises a novel "Story-Writing-Coding" engine we have developed. We compare novice (children's) processes of coding an animated story with that of experts (college students) and evaluate the differences using four measures based on the progressive coding of a complete program. We also analyse the use of novice (children's) computational thinking in this coding process. This research is set against a backdrop of approaches to teaching programming and concepts of computational thinking in recent educational literature.

Introduction

Computer Science is now being taught in primary schools in England as a statutory requirement of the primary National Curriculum Department for Education (DfE, 2013a). The aims of this primary programme are "to use computational thinking and creativity to change the world" and to use children's knowledge of computing to "create programs, systems and a range of content" (DfE, 2013a, p.1). Currently children aged 5-11 (Kindergarten – 5th grade) in England are engaged in learning about programming and computational thinking (CT). In the US, early in 2016 President Obama launched the initiative 'Computer Science for all' with the aim of empowering US students, from Kindergarten through high school, to learn computer science and to be equipped with the computational thinking (CT) skills needed to be creators and not just consumers in the digital economy. Later that year, the Association for Computing Machinery with partners released the K-12 Computer Science Framework intended to inform the development of both standards and

curriculum (K-12 CSF, 2016). Like the English primary programmes of study (DfE, 2013a), there is core reference to both computational thinking (CT) and programming.

However, within computer science education research there is a vigorous discussion on how best to approach the teaching of programming and of the nature of CT itself. In terms of programming, there is gathering momentum to use non-textual, block-based tools such as Scratch (Morris, Uppal & Wells, 2017), as opposed to traditional text-based approaches taught at university and used professionally. In terms of CT, despite the appearance of this term in statutory requirements or guidance documents, there seems to be an ongoing discussion on what this means, and how to implement it in the curriculum.

This paper reports on a study that forms part of a large research project where we are investigating how to teach primary school children to program in a text-based language (Java) typically taught to undergraduate/college students. The programming environment used is our novel Story-Writing-Coding engine, where children code an animated story. Previous work [citation of authors suppressed] has focused on the creation of meaning using code. This paper focuses on a comparison of the code written by the novice children with that written by experts, where the expert comparison group is taken from our undergraduate students. The rationale for doing this is detailed in the Methodology section. We have two research questions: (i) How can we measure the difference between children's and experts' processes of coding, and what do these differences reveal? (ii) Is there any evidence for CT in children's programs?

This paper is structured as follows: In the following section we review the literature on computational thinking and programming. The next section provides an overview of our animation engine, how it was designed to support both programming and CT. This is followed by an overview of our research methodology, that identifies participants and processes, then discusses instruments designed to measure program development and assessment of evidence of CT. The results section provides an analysis and comparison of 18 programs written by children with 13 written by experts

according to our instruments. In the closing discussion section we provide answers to our two research questions and attempt to tease out possible implications for teachers.

Computational Thinking, and Programming

Computational Thinking

Computational Thinking (CT) proposed by Jeanette Wing in 2006, revised in 2008 (Wing, 2008) and more recently revised in 2011 (Wing, 2011) has entered the school curriculum both at secondary and primary level in England (DfE, 2013a; DfE, 2013b). It is included in the College Board Advanced Placement course (CBAP, 2016), and most recently is central to the US K-12 Computer Science Framework (K-12 CSF, 2016).

Wing's 2006 proposal made several significant points, first that "Computational thinking builds on the power and limits of computing processes, whether they are executed by a human or by a machine" (Wing, 2006, p.33). She goes on to say that "It represents a universal attitude and skill set everyone, not just computer scientists, would be eager to learn and use" (Wing, 2006, p.33), and that it "involves solving problems, designing systems and understanding human behaviour, by drawing on the concepts fundamental to computer science" (Wing, 2006, p.33). The clear suggestion here is that the existence of computers has provided humans with a novel mode of thinking, a new epistemology to be applied to many disciplines, and therefore cross-curriculum. Later she crystallises her initial ideas, "The essence of computational thinking is *abstraction*" (Wing, 2008, p.3717) and asserts that this will have benefit to education in general; "Computational thinking is not just or all about computer science. The educational benefits of being able to think computationally – starting with the use of abstraction – enhance and reinforce intellectual skills, and thus can be transferred to any domain" (Wing, 2011, p.4). Grover and Pea (2013) provide a comprehensive overview of CT, and how it resonated with educators, educational researchers and policy makers. They highlight the struggle to agree on a definition of CT and point to the Royal Society (2012) as offering a useful definition: "Computational thinking is the process of recognising aspects of computation in the world

that surrounds us, and applying tools and techniques from Computer Science to understand and reason about natural and artificial systems and processes" (Royal Society, 2012, p.29).

However, there are some warnings. Peter Denning warns that "the computational thinking movement reinforces a narrow view of the field [computer science]" (Denning, 2009, p.28). He advocates that it would be unwise to propose that CT is a defining element of computer science, since it ignores the history of thinking in computer science as well as in all the sciences. Denning reminds us that "Computation is more fundamental than computational thinking. For this reason alone, computational thinking seems like an inadequate characterisation of computer science" (Denning, 2009, p.30). The most recent attempt to define CT is found in Yadav, Stephenson and Hong, (2017) who propose "… computational thinking is a set of problem-solving thought processes derived from computer science but applicable in any domain" (Yadav et al., 2017, p. 56).

Let us now turn to the response to CT by educationalists and policy-makers. Today's K-12 education is a complex and politicised system, where subject content, ideologies and pedagogies seem to be in competition. At the same time the classroom is subject to high levels of formal expectations and scrutiny. Against this backdrop there appears to be some divergent "philosophies" of CT. First, there appears to be a clear aim to separate computational thinking from programming, both within interpretations of the English National Curriculum (Computing At School, 2015; Selby, Dorling, & Woollard, 2014) as well as in the 'computer science Unplugged' effort that introduces computing concepts without the use of a computer. These unplugged approaches may serve to apply computational thinking beyond computer science to a variety of other disciplines (Bundy, 2007). However, it is suggested that these, "while providing valuable introductory activities for exposing children to the nature of computer science, may be keeping learners from the crucial computational experience involved in CT's common practice" (Grover & Pea, 2013, p.40). This warning reminds us not to forget the origins of CT in programming.

The definition of CT in the K-12 Computer Science Framework (2016) is interesting, in both the debugging and creating components; there is *explicit* mention of creating computational artefacts

unlike in the interpretations of the English National Curriculum. This emphasises that CT is at the heart of computer science, “The most effective context and approach for developing computational thinking is learning computer science, they are intrinsically connected” (K-12 CSF, 2016, p69). Also, this framework includes core practices that extend CT; communicating about computing, collaborating around computing and fostering an inclusive computing culture (K12 CSF, 2016). These align with Barr and Stephenson’s (2011) suggestions of negotiation and consensus building; indeed they highlight the need for teamwork, and include both abstraction and decomposition under this heading. Substantial work to establish a comprehensive strategy for getting CT into schools has been made by the Scalable Game Design Initiative (SGDI) (Repenning, Webb & Ioannidou, 2010) based on the establishment of CT patterns (Ioannidou, 2011), and proposing an instrument for the automatic recognition of CT using these patterns (Koh, Basawapatna, Bennet & Repenning, 2010).

The literature reviewed suggests a ranking of CT factors: abstraction is the favourite, in agreement with Wing (2008) and Grover and Pea (2013); followed by algorithmic thinking, decomposition, generalisation, testing, and debugging in equal second place; followed by logical thinking.

Programming

The literature uses varied vocabulary to distinguish between text-based and block-based approaches. The latter are often referred to as “visual” or “graphical” programming environments, which is confusing since many text-based approaches (such as ours) deliver visual/graphical output. To avoid confusion we shall refer to these contrasting approaches as “block syntax” or “text syntax” following Stead and Blackwell (2014). There are many languages available for teaching primary school children to code, most use block syntax. Extensively used outside the classroom, they are now being used within the English computing curriculum. An excellent overview is found in Morris et al., (2017) with suggestions of suitable languages and approaches for all stages of primary education and also at the transition into lower-secondary schools.

The K-12 CSF explicitly refers to programming, “Computers also require people to express their thinking in a formal structure such as a programming language” (K-12 CSF, 2016, p.69). By “formal

structure" we think of flow diagrams and pseudo-code. Yet programming is more than designing algorithms a computer can execute, "Creating a program allows people to externalise their thoughts in a form that can be manipulated and scrutinized. Programming allows students to think about their thinking" (K-12 CSF, 2016, p.69). Again this strengthens the link between teaching CT and programming.

The English National Curriculum for computing, in addition to aspects of CT, aims that all pupils should "have repeated practical experience of writing computer programs …" and specifies the subject content for Key Stage 1 (Kindergarten and 1st Grade) and Key Stage 2 (2nd to 5th Grade) (DfE, 2013a, p.1). Comparing these with the K-12 CSF programming concepts, we find the former are rather abstract, specifying "what" while the latter are more concrete, often specifying "how". For example, the former specifies "Create and debug simple programs" (DfE, 2013a, p.2), while the latter in reference to programs states "Sprites can be moved and turned", "… drawing a shape or moving a character across a screen" (K-12 CSF, 2016, p.96).

However, commenting on the National Curriculum programming requirement, some UK computer science educationalists note that "It is deliberately not mandated that this learning should take place on an actual computer" (Brown, Sentence, Crick & Humphreys, 2013, p.9) and they go on to suggest the computer science unplugged "… can cover the requirements of the curriculum without programming a physical computer". More recently Yadav et al. (2017) seem to agree, that while CT is "deeply connected to the activity of programming, it is not essential to teach programming as part of a pre-service computational thinking approach" (Yadav et al., 2017, p.58). However, other researchers are quite explicit about the value of coding, "an introduction to programming is not necessarily a precursor to teaching algorithmic thinking, but rather provides the very means to teach algorithmic thinking", (Hromkovic, Kohn, Komm & Serafini, 2017, p.1); here programming refers to Logo and Python.

Block syntax environments for young learners are widely endorsed by the Computers At Schools movement (CAS, 2015). This is justified by research into the detailed links between block syntax

(Scratch) and CT focusing on both concepts and practices of CT (Brennan & Resnick, 2012). Other educators suggest that starting with block syntax then progressing to text syntax may be appropriate for primary children (Morris et al., 2017) and cite Logo as an appropriate tool. It has been suggested that Scratch is a good first language to learn and Scratch will help transitioning to a text syntax language later on (Dorling & White, 2015). While this seems a reasonable assertion, there is little evidence to support it. A more rigorous study suggests the opposite; some 120 students aged 15-16 (10th Grade) participated, a group of 44 had prior experience of Scratch and a group of 76 did not. The results showed no significant differences between the groups for the concept of variables and conditional execution, yet significant differences in using repetition in favour of those with prior experience of Scratch, (Armoni, Meerbaum-Salant & Ben-Ari, 2015).

There is also evidence that block syntax can induce "bad habits", (Meerbaum-Salant, Armoni & Ben-Ari, 2011). This research looked at 14-15 year olds (9th grade) learning Scratch without instructional materials. It reported "bad habits" acquired, those which did not encourage designing algorithms or using programming constructs (selection, iteration) and showed how these were a consequence of the drag-and-drop nature of Scratch. Children first collected all blocks they thought appropriate and then combined them into several scripts without any planning or thought. One sophistication of Scratch is that it can run many scripts concurrently. Meerbaum-Salant et al., (2011) showed how this feature, together with the block-collection behaviour, led to incorrectly structured code (which nevertheless ran). Effectively, children were 'misusing' the power of Scratch's concurrency. This approach to coding is called 'bricolage' (tinkering) where the coder assembles code by trial and error without planning. This is confirmed by more recent research (Rose, 2016). Advice to teachers in the publication "Quickstart Computing" (CAS, 2015) suggests that Scratch and Kodu can make it seem unnecessary to go through the planning stage of writing a program. It goes on to suggest that it is good practice for pupils to write down the algorithms for a program, in the form of rough jottings, storyboard, pseudocode or flow charts.

There has been one previous attempt to link storytelling and coding; “Storytelling Alice” uses block-syntax to produce 3D animations. It was created for middle school girls, to address the under-representation of women in computer science (Kelleher, 2006; Kelleher & Pausch, 2007). The “Greenfoot” environment provides a Java-based text syntax approach aimed at children aged 14 and above (Kolling, 2010), while Scratch aims at 8-16 year olds and Alice 12-19. Here, users drop objects into a world using block syntax then code the objects’ methods (in templates automatically provided) using the full Java language, including many of the characteristics of Object-oriented Programming.

With the exception of Greenfoot, these approaches do not provide learners with experience of syntactic features of conventional languages which they ultimately must learn; text syntax is replaced by coloured jigsaw pieces. The “Drawbridge” approach presents a sophisticated combination of block and text syntax, and with an easier user interface (Stead, 2016; Stead & Blackwell, 2014). Block syntax can be viewed side-by side with text syntax (JavaScript) with simultaneous update. In this way children can move from block to text syntax. Robust trials with children aged 11-12 (6th grade) have shown that starting with blocks rather than text does improve understanding of text syntax, (Stead & Blackwell, 2014).

Computer games form a useful platform to learn programming; games can balance challenge against expectations and achievement. “Lightbot” is such a game proposed for primary schools (see Morris et al., 2017) where the objective is to switch on all lights in a level using fixed commands. As the game levels progress, functions and selection statements are introduced. Research by Gouws, Bradshaw and Wentworth (2013), suggests that Lightbot is useful in developing CT; unlike the potential “bricolage” approach using Scratch, Lightbot enforces a sequential mode of program design.

The principle of “low floor, high ceiling” to guide the development of programming environments is well known since the days of Logo (Flannery et al., 2013) and is capitalised in computer game programming, to a professional level. Here, the Scalable Game Design Initiative (Ioannidou, 2011;

Repenning et al., 2010), suggests that all CT tools should have a “low threshold” and a “high ceiling”, so that novices can rapidly create a working game, but allow the creation of a game with sophisticated behaviour. Many programming tools suitable for K-12 education fit this requirement including Scratch, Alice, Kodu, and Greenfoot, (Morris et al., 2017). Perhaps of the environments mentioned above, Greenfoot has a relatively high floor while Drawbridge and Scratch have a low floor. In our experience Storytelling Alice, despite its block syntax, has a relatively high floor.

Overview of the Animation Engine

This section outlines various “affordances” we have built into the engine to support meaningful Story-Writing-Coding. Programming affordances are fundamental, we expect coders to engage with these to create meaningful programs. CT affordances are more abstract and are intended for instructors to select from when appropriate.

Programming Affordances

Our engine is based on the Java language where children code using Java syntax, for example the line of code **grog.flyto(myrobin);** which makes the character “grog” fly to the goal “my robin” uses the object-based syntax, where “grog” is the *object*, “flyto” is the *method* and “myrobin” is a *parameter.* This object-based approach is a simplification of the full object-oriented approach to programming that allows additional *classes* to be created, e.g., a new sort of object such as “background” with methods different from those of characters. This would require users to work with additional code entry boxes (and at a higher cognitive level) that was judged too difficult for children of this age.

Coders are able to add scenery, props and characters to a canvas. Props and characters can move and characters can display emotions. Character methods are based upon Halliday’s “Systemic Functional Grammar”, that aims to teach how to get meaning into a written text (Halliday, 2004), in this case through code. Halliday (2004) proposes a classification of linguistic “processes” (verbs) into classes that encompass e.g., movement, speaking, thinking and feeling. The classes we have

implemented, together with the code methods, and examples of story vocabulary are shown in Table 1.

-- Table 1 about here –

The second designed affordance allows coding either synchronous or asynchronous character behaviour. This is not trivial since users write all code as a sequence of statements in a single code-text input box. This is achieved by programming using “tuples” of statements, e.g. the code for two characters should be arranged in pairs as shown in the code snippet in Table 2,

-- Table 2 about here –

On the left, pip and grog jump at the same time, while on the right pip jumps first followed by grog jumping. We also provide polymorphic forms of most methods with decreasing abstraction. For example this **jump();** is called without parameters using a built-in jump height, while **jump(50);** is less abstract, allowing the jump height to be specified. A further reduction in abstraction **jump(50,4);** allows the time taken for the jump (4 seconds) to be included.

The user interface is shown in Figure 1. On the left is the code entry box where the coder can add one, many or no lines of code; pressing the “run” button compiles the code and displays the associated animation on the canvas to the right. This provides immediate feedback of the effect of the code written; the user can continue to code, modify existing code or correct errors. This rapid feedback is important and motivates the learner to maintain momentum. Pressing “run” also logs a rich data file containing the code, and details of any compilation errors at that point. We are therefore able to track *changes* in the code. This is crucial since it provides more information about code development and CT than can be gleaned from the final program, as noted by Brennan and Resnick (2012).

-- Figure 1 about here –

Computational Thinking Affordances

The engine has been designed to teach programming, and its support for CT must be viewed in this light. We suggest a hierarchy of coding activities and we align the CT factors against this hierarchy. We view this hierarchy as guidance for practitioners to teach elements of CT grounded in programming activities.

At the top is the most abstract of all coding, composing the story itself. Here we expect to see *decomposition* where children organise their code into blocks that could represent the various phases in an unfolding story (setting the scene, some conflict, resolution and ending). We also expect to see *abstraction* here, since there will be clauses that cannot be coded (e.g. expressions of cause or purpose). In addition, there is another dimension of thinking related to writing stories in general; these are story-patterns referred to as "narrative schemas" (Bruner, 1991). Moving down a level we expect to see *patterns* made from tuples. These could be viewed as *decomposition* into tuples or equally well *abstractions* focussing on pattern behaviour rather than the underlying tuples. Below patterns we find code tuples that form the realisation of the *abstract* concepts of concurrency and sequentiality. Towards the base of the pyramid we find *abstraction* again in the use of polymorphic method forms, and *algorithmic thinking* in the use of selection and iteration constructs. Finally, we expect to see *logical thinking* in the changes made to code (indicated by the "*purposeful coding effort"* measure, see below) to obtain the author's desired effect. Moving down this hierarchy, the CT skills become less abstract. One important level for our engine, in this hierarchy, is *patterns*. The idea of using patterns to reflect CT was proposed by Ioannidou (2011) who, in the context of computer games produced by middle-school children, classified the observed behaviour of game objects into patterns such as "collision", "absorption", "generation", "diffusion". Our basic patterns comprise organised movement of characters in space-time realised through sequences of "tuples" producing behaviour such as characters *meeting up, chasing, conversing, observing.*

Research Methodology

Participants and Procedures

This study used an “exploratory data analysis” design where no initial hypotheses were stated; data was collected from two groups where we sought to analyse the difference in coding between the groups. Group 1 comprised 18 primary school children Yrs 3 to 6 (Grades 2- 5) from a rural and an inner-city school and group 2, the “comparison” group comprised 13 “expert programmers”, final year Computing students at the University [name suppressed]. The use of undergraduate/college students as a comparison group may appear unusual, yet we propose they provided an ideal comparison group since we are comparing novice and expert programmers. None of the children had prior experience of text-based programming, 6 of them indicated having used Scratch. All undergraduate students had experience of coding using at least three text-based languages including Java, but they had not met the engine before. We therefore expected them to have stable and robust mental models of coding, including knowledge of programming constructs, program composition and error correction strategies. Both groups were given the same task, to code an animated story, using the same instructional materials; therefore, our comparison would reveal differences in coding processes using this particular engine; a focused and objective comparison of the two groups.

The study comprised two phases; during the first instructional phase participants were introduced to the engine affordances. They were shown how to add an item of scenery, a character and to make the character move, see the code snippet in Table 3.

-- Table 3 about here --

Then they were asked to experiment with the character methods shown in Table 1. Following that, programming constructs and the use of tuples were taught directly. This phase lasted two hours and was followed by the second phase where participants were asked to independently code a story, either a known or made-up story or else letting the story emerge from coding, with help being available. The second phase followed one week later and lasted one hour.

Programming Measurements

Primary data comprised extensive records of the participants' coding activities, automatically recorded by the engine, that contained time-shots of code (when the "run" button was pressed) including errors reported by the compiler. All records were subject to manual *post-hoc* analysis. For each participant we extracted the number of lines of code written, the time taken, the number of corrected errors and the number of tuples used. We introduced a new measure, the *purposeful coding effort* (PCE) that aims to capture the purpose behind changing code. This number was obtained manually for each record by incrementing its value for the following changes between each run: adding or deleting a line of code, rearranging lines of code, changing the method for a character, changing a character, changing a parameter. Error correction was not included. A value PCE = 1 is the baseline and corresponds to simply adding lines of code. Values greater than 1 indicate the number of changes made to existing code, therefore this measure indicates the amount of "purpose" in interacting with the developing program.

All measures were normalised relative to the total lines in the finished program, yielding PCE/line, tuples/line, errors-corrected/line and lines-written/minute; these were then subjected to non-parametric statistical tests. We first ran an exploratory analysis including the Shapiro-Wilk test that indicated that all test data were not-normally distributed (e.g. W = 0.88, p = .03). We then ran the two-sample Mann-Whitney Wilcoxon Test for independent variables on each measure, and the significance of any differences in the median, and also the effect size between the two groups on each measure were calculated. Since this involved running multiple tests on the same data set the familywise error would increase beyond the 0.05 significance level. To mitigate against this we applied Holm's variant of the Bonferroni correction (Holm, 1979), where the p-values for each test are first ranked in increasing order, and then an adjusted alpha value for each test is calculated as 0.05/rank. To report the effect size we calculate Pearson's r according to the expression $r = z/\sqrt{N}$ where z is the z-score and N the total number of observations (Rosenthal, 1991, p.19). The resulting effect sizes lie between 0 and 1, where as a rule of thumb $r = 0.10$ is a small effect, $r = 0.$30 is a

medium effect and *r = 0.*50 is a large effect (Cohen, 1992). All calculations were done using packages in the "R"-language.

Computational Thinking Measurements

To answer our second research question, we restricted measuring CT to the children. Working down the hierarchy presented above, we first asked the children to demonstrate their coded story and to recount their story. From this we judged whether a meaningful story had been constructed. We then reviewed their code looking for use of *decomposition*. The animation was reviewed and we looked for evidence of patterns in the animation and traced any pattern back to the program where we looked for evidence of patterns having been made from tuples (*abstraction, patterns*). Evidence of *abstraction* shown by polymorphic method calls was sought and for records with PCE > 1 we looked for evidence of *logical reasoning*. Finally we looked for examples of *algorithmic thinking* through the use of programming constructs.

Results of Analysis

Programming

Results of the Mann-Whitney Wilcoxon Tests are presented in Table 4.

-- Table 4 about here –

First the lines/min measure for the children (Mdn = 0.53) differed significantly from students (Mdn = 1.13), W = 38.5, p = .003, r = 0.55; children are writing code at a lower rate than students, with a large effect size. This is expected and reveals the differences in mechanical (keyboard) skills between the groups as well as general cognitive development. Second, the PCE/line measure for the children (Mdn = 1.07) differed significantly from students (Mdn = 1.57), W = 31, p = .0009, r = 0.61 with a very large effect size. We find that 8/18 children have PCE/line = 1; they are coding a story by simply building a sequence of statements. Those with PCE/line > 1 are either, changing parameters, changing methods on characters, deleting or moving lines of code. For example child 9 (PCE = 1.61) made a series of parameter changes on scenery to adjust the cohesion of the scene. Child 18 (PCE =

1.54) changed the target location for a character's movement and changed a method to control a character's size. Child 5 (PCE = 1.41) changed character method parameters to change their appearance, see Table 5.

-- Table 5 about here --

Third, the error corrected/line measure for the children (Mdn = 0.11) did not differ significantly from students (Mdn = 0.14), W = 66, p = .29, r = 0.19, and the effect size is small. This is an interesting result; there seems to be little difference between the rate of introduction of errors between groups. Inspection of the most frequent errors made by both groups show some similarities and differences. Both groups miss-spell key words and forget the ";" statement terminator. Children often use incorrect method syntax, and students often use an incorrect object. Finally the tuples/line measure for the children (Mdn = 0.07) differed significantly from students (Mdn = 0.0), W = 171.5, p = .023, r = 0.44, showing a large effect size. Children have learned synchronisation using tuples and deploy this to good effect, see Child 10 in Table 5. Inspection of students' code revealed they had developed a heuristic approach to synchronisation, e.g., having discovered that adding an object synchronises code up to that point.

In summary, we have found that students outperform children in both the speed of writing code and in changing code with a purpose in mind. Children outperform students in the use of tuples, and there is little difference in correcting errors.

Computational Thinking

We found most children clearly organised their code into "blocks", e.g. to set up the scene, to have character interaction, and to bring the story to its end (*decomposition*). Three children went beyond this and created their own methods to separate out parts of their stories. We found evidence for patterns (*abstraction*) where children combined tuples to great effect. Most children used patterns as expected, characters would *meet up*, they would *have conversations*, but some children coded more complex patterns. One child created a *dance routine* pattern, where two characters executed

synchronous code to obtain a movement pattern that had mirror-symmetry in the horizontal direction through judicious selection of parameters; this is clear evidence of *logical thinking*, see Table 6

-- Table 6 about here --

Children made the correct use of polymorphic forms, e.g. using either **grog.flyto(40,20);** or **grog.flyto(pip);** depending on the current layout of the scene, and they chose suitable method parameters, though boys would occasionally exaggerate their values to obtain strange effects such as high rates of spinning. Here is evidence of *abstraction*. There is also evidence of what we suggest is "higher-order" abstraction. Consider this line of story-text. "Grog flies to see Pip *because they are friends*". The second clause (of reason) cannot be coded, the child who wrote this has abstracted out that story-text which cannot be coded, but which they feel important for their story. Concerning *algorithmic thinking*, even those children who did not use tuples were able to correctly sequence lines of code to produce their desired animation. Only two children transferred their learning of programming constructs; one child used iteration to assemble a forest of trees, a second child used iteration to generate a sequence of actions to create an excited character. While this is a little disappointing, it may be that it exceeds their cognitive capacity, given the range of engine affordances available, or simply they did not need to use these constructs. We found no evidence of 'bricolage', looking at children with PCE > 1 we found they were continually making purposeful choices, adding and changing code. They were clearly drawing down their experience of story-writing from literacy classes and from their reading experiences. In summary there is evidence of types of thinking going on, and much of this can be described as computational thinking.

Discussion

In answer to our first research question "How can we measure the difference between children's and experts' processes of coding, and what do these differences reveal?", the four measures we have devised report different aspects of children's coding process and how this differs from experts. While experts code faster and are more adept at modifying code for a purpose, the children make

better use of tuples to make the abstract concept of synchronisation concrete and straightforward. Unexpectedly we found there was no difference in the rates of error correction. The purposeful coding effort (PCE) measure revealed useful information about how children compose their programs. A future study will investigate the use of this measure over groups of children of different ages in K-12 to explore age-related differences. In answer to our second research question "Is there any evidence for CT in children's programs?" we have found clear evidence of children using the concepts of *abstraction, decomposition, logical thinking and patterns*. We are tempted to speculate that the use of patterns is related to the Story-Writing-Coding context.

There are of course limitations to this study. The sample size was small, however this is mitigated by the use of the appropriate statistical tests. The lack of use of selection and iteration (*algorithmic thinking*) is a concern and may point to a limitation of Story-Writing-Coding, or at least of the engine in its current form. While we are confident about our analysis of CT as presented here, we have some reservations about the CT factors identified in the literature. Future research will focus on *abstraction, logical thinking* and *patterns* that pick up various elements discussed in this paper.

This research leads us to formulate suggestions for the practitioner. First we encourage primary teachers to consider using a text-based language in their teaching, especially platforms that produce graphical output. Second we encourage them to monitor the *development* of children's programs over time, using a measure such as our PCE. Third we encourage them to critically evaluate concepts within CT, and to teach these linked closely to programming. Fourth we encourage teachers to adopt our Story-Writing-Coding approach, since it taps into the inherent desire and ability of children to tell stories. Finally we invite researchers to consider using students as a comparison group to evaluate children's progress. The authors are willing to share the engine and a range of teaching resources suitable for K-12 educators, and we will also be pleased to receive requests for collaboration, please contact the corresponding author.

References

Armoni, M., Meerbaum-Salant, O., & Ben-Ari, M. (2015). From Scratch to "Real" Programming. *ACM Transactions on Computing Education*, *14*(4), 25:1 – 25:15.

Barr, V., & Stephenson, C. (2011). Bringing Computational Thinking to K-12: What is Involved and what is the role of the computer science education community? *ACM Inroads*, *2*(1), 44-54. doi:10.1145/1929887.1929905.

Brown, N. N. N., Sentance, S., Crick, T., & Humphreys, S. (2013). Restart: The resurgence of Computer Science in UK schools. *ACM Transactions on Computing Education, 1*(1), 1:1-1:22.

Brennan, K., Resnick, M. (2012). New frameworks for studying and assessing the development of computational thinking. Paper presented at the American Education Researcher Association, Vancouver, Canada. Retrieved from http://web.media.mit.edu/~kbrennan/files/Brennan_Resnick_AERA2012_CT.pdf

Bruner, J. (1991). The narrative construction of reality. *Critical Inquiry, 18*, 1-21.

Bundy, A. (2007). Computational Thinking is Pervasive. *Journal of Scientific and Practical Computing, 1*(2), 67-69.

Cohen, J. (1992). A power primer. *Psychological Bulletin, 112*(1), 155-159.

Computing At School [CAS] (2015). Quickstart Computing. A CPD toolkit for primary teachers. Retrieved from https://www.computingatschool.org.uk/quickstart#v1

College Board Advisement Placement [CBAP] (2016). AP Computer Science Principles. Retrieved from https://advancesinap.collegeboard.org/stem/computer-science-principles

Denning, P. J. (2009). The profession of IT. Beyond Computational Thinking. *Communications of the ACM, 52*(6) 28-30. doi:10.1145/1516046.1516054.

Department for Education [DfE] (2013a). *Computing programmes of study: key stages 1 and 2*. London: HMSO.

Department for Education [DfE] (2013b). *Computing programmes of study: key stages 3 and 4*. London: HMSO.

Dorling, M., & White, D. (2015). *Scratch: A Way to Logo and Python*. Presented at the SIGCSE Conference, Kansas City, MO, USA, 2015, doi:10.1145/2676723.2677256.

Flannery, L. P., Silverman, B., Kazakoff, E. R., Bers, M. U., Bonta, P., & Resnick, M. (2013). Designing ScratchJr: Support for early childhood learning through computer programming. *Proceedings of the 12th International Conference on Interaction design and Children*, ACM, New York, NY, USA.

Grover, S., & Pea, R., (2013). Computational Thinking in K-12: A review of the state of the field. *Educational Researcher, 42*(1), 38-43. doi:10.3102/0013189X12463051.

Gouws, L. A., Bradshaw, K., & Wentworth, P. (2013). Computational thinking in educational activities: an evaluation of the educational game light-bit. *Proceedings of the 18th ACM conference on Innovation and technology in computer science education,* ACM, Canterbury, England, UK. doi:10.1145/2462476.2466518.

Halliday, M. A. K. (2004). Halliday's Introduction to Functional Grammar. Oxford: Routledge.

Holm, S. (1979). A simple sequentially rejective multiple test procedure. *Scandinavian Journal of Statistics, 6*(2), 65-70.

Hromkovic, J., Kohn, T., Komm, D., & Serafini, G. (2017). Algorithmic Thinking from the Start. *Bulletin of the European Association for Theoretical Computer Science*, *121*, February 2017. Retrieved from http://bulletin.eatcs.org/index.php/beatcs/article/view/478

Ioannidou, A. (2011). Computational thinking patterns. Paper presented at the 2011 annual meeting of the *American Educational Research Association*.

Kelleher, C. (2006). *Motivating Programming: Using Storytelling to Make Computer Programming Attractive to More Middle School Girls* (Doctoral dissertation). Carnegie Mellon University. Retrieved from http://www.cs.cmu.edu/~caitlin/kelleherThesis_CSD.pdf

Kelleher, C., & Pausch, R. (2007). Using Storytelling to Motivate Programming. *Communications of the ACM, 50*(7), 59-64.

Koh, H. K., Basawapatna, A., Bennett, V., & Repenning, A. (2010). Towards the automatic recognition of Computational Thinking for adaptive visual language learning. IEEE *Symposium of Visual Languages and Human-Centric Computing*, Leganes, Spain 2010, 59-66. doi:10.1109/VLHCC.2010.17

Kolling, M. (2010). The Greenfoot programming environment. *ACM Transactions on Computing Education*, *10*(4), 14:1-14:21.

K-12 Computer Science Framework [CFS] (2016). *K-12 Computer Science Framework*. Retrieved from http://www.k12cs.org

Meerbaum-Salant, O., Armoni. M., & Ben-Ari, M (2011). *Habits of Programming in Scratch*. Presented at the ITiCSE conference June, 2011, Darmstadt, Germany.

Morris, D., Uppal, G., and Wells, D. (2017). *Teaching Computational Thinking and Coding in Primary Schools.* Thousand Oaks, CA: Sage Publications.

Repenning, A., Webb, D. C., Koh, K. H., Nickerson, H., Miller, S. B., Brand, C., Horses, I. H. M. (2015) Scalable game design: A strategy to bring systemic computer science education to schools through game design and simulation creation. *ACM Transactions on Computing Education*, *15(*2), 11:1-11:31. doi:10.1145/2700517

Repenning, A., Webb, D., & Ioannidou, A. (2010). Scalable game design and the development of a checklist for getting computational thinking into public schools. *Proceedings of the 41st ACM technical symposium of Computer science education,* Milwaukee, WI, USA March 2010.

Rose, S. (2016). Bricolage programming and problem solving ability in young children: An exploratory study. In 10th European Conference on Games Based Learning, University of the West of Scotland, Paisley October 2016. Retrieved from

http://shura.shu.ac.uk/12649/3/Rose%20Bricolage%20programming%20problem%20solving%20ability.pdf

Rosenthal, R. (1991) *Meta-analytic procedures for social research* (2nd ed.) Newbury Park, CA: Sage.

Royal Society, (2012). Shut down or restart: The way forward for computing in UK schools. Retrieved from http://royalsociety.org/education/policy/computing-in-schools/report/

Selby, C., Dorling, M., & Woollard, J. (2014). Evidence of assessing computational thinking. Retrieved from https://eprints.soton.ac.uk/372409/1/372409EvidAssessCT.pdf

Stead, A. G. (2016). *Using multiple representations to develop notational expertise in programming.* Technical report based on Ph.D Dissertation, University of Cambridge, UK. Technical report number UCAM-CL-TR-890 retrieved from http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-table.html

Stead, A. G., & Blackwell, A. (2014). *Learning syntax as notational expertise when using Drawbridge*. In B. du Boulay and J. Good (Eds), Psychology of Programming Interest Group Annual Conference (2014). Retrieved from http://www.ppig.org

Wing, J. M. (2006). Computational Thinking. *Communications of the ACM, 49*(3), 33-35.

Wing, J. M. (2008). Computational thinking and thinking about computing. *Phil. Trans. R. Soc. A, 366,* 3717-3725. doi:10.1098/rsta.2008/0118

Wing, J. M. (2011). Research Notebook: computational Thinking – What and Why? Retrieved from https://www.cs.cm/link

Yadav, A., Stephenson, C., & Hong, H. (2017). Computational Thinking for Teacher Education. *Communications of the ACM, 60*(4). 55-62. doi:10.1145/2994591.

Table 1. A selection of programming methods derived from Systemic Functional Grammar.

Movement At	Movement To	Appearance	Possession	Talking and Thinking	Emotions

pip.jump();
pip.jump(50);
pip.spin();
pip.flip();
	pip.flyto(10,10);
	pip.flyto(grog);
	pip.leapto(10,10);
	pip.leapto(grog);
	pip.walkto(50);
	pip.runto(50);
		pip.hide();
		pip.show();
		pip.grow(1.5);
		pip.shrink(0.5);
			pip.pickup(grog);
			pip.putdown(grog);
				pip.says(“Hi”);
				pip.thinks(“No”);
					pip.feels(happy);
					pip.is(sad);

Table 2. Tuples (here pairs) of code statements used to achieve synchronisation of actions.

Both pip and grog jump together	Pip jumps then grog departs

pip.jump();
grog.jump();
	pip.jump();
	grog.rest();
	pip.rest();
	grog.flyto(100,10);

Table 3. Lines of code explained to participants before they experimented with other statements.

```
add(bigtree,70,10);
add(grog,20,15);
grog.jump(30);
```

Table 4. Results of statistical analysis for children (child) and students (stud).

Measure	Median (child)	Mean (child)	Median (stud)	Mean (stud)	W	*p*	rank	α' (0.05/rank)	*r* (effect size)
PCE/line	1.07	1.14	1.57	1.54	31	0.0009	4	0.0125	0.61
Lines/min	0.53	0.49	1.13	1.28	38.5	0.003	3	0.017	0.55
Tuples/line	0.07	0.11	0.0	0.02	171.5	0.023	2	0.025	0.44
Error Correction /line	0.11	0.12	0.14	0.16	90.5	0.29	1	0.05	0.19

Table 5. Examples of children's changes to code during development.

	Child 18	Child 9	Child 5	Child 10
	Changing Methods	Changing parameters	Changing parameters	Using tuples

Initial code	myant.grow(1.5);	add(bigtree,10,50);
		add(bigtree,20,30);
			myant.grow(1.5);
			myant.rest();
			myant.shrink(0.5);
				pip.flyto(myshell);
				grog.rest();

First change	myant.shrink(0.5);	add(bigtree,50,10);
		add(bigtree,20,40);
			myant.grow(4.5);
			myant.rest();
			myant.shrink(0.1);
				pip.flyto(myshell);
				grog.rest();
				pip.pickup(myshell);
				grog.rest();

Second change		add(bigtree,50,10);
		add(bigtree,30,30);
			myant.grow(4.5);
			myant.rest();
			myant.shrink(0.01);
				pip.flyto(myshell);
				grog.rest();
				pip.pickup(myshell);
				grog.rest();
				pip.rest();
				grog.flyto(mystar);

Third Change		add(bigtree,50,10); add(bigtree,40,30);		

Table 6. Code expressing logical thinking, obtaining symmetrical movements.

Code for grog	grog.jump(10);
	grog.spin(5);
	grog.rest();
	grog.walkto(30);
	grog.runto(40);
	grog.hopto(5);

Code for pip	pip.jump(10);
	pip.spin(5);
	pip.rest();
	pip.walkto(50);
	pip.runto(40);
	pip.hopto(80);

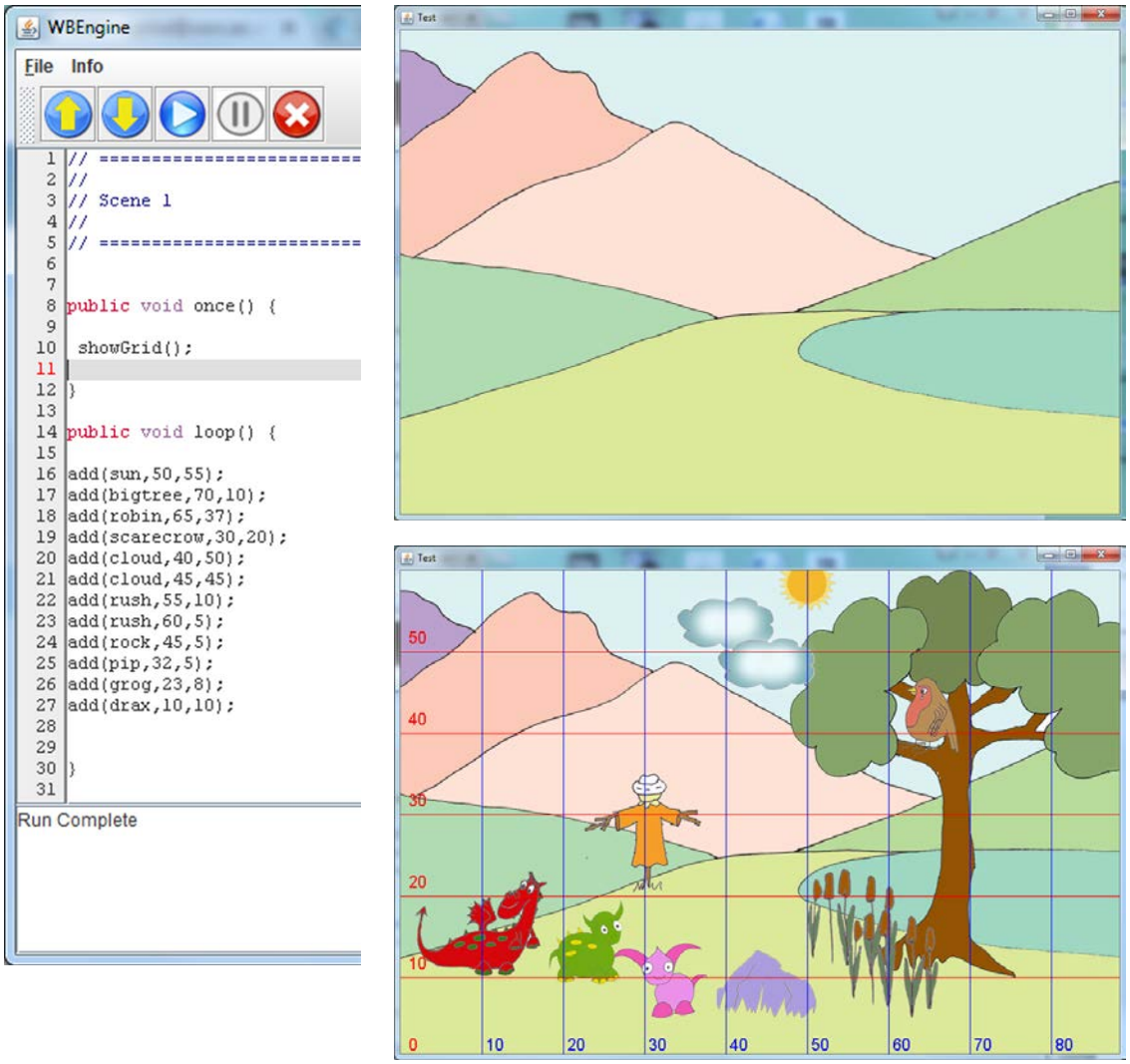

Figure 1. The user interface showing the code entry box on the left and the canvas on the right. The user sees a single canvas; this is repeated here to show the effect of the code execution.