

Code ABC MOOC for Math Teachers

Pia Niemelä¹, Tiina Partanen², Linda Mannila³, Timo Poranen⁴, and Hannu-Matti Järvinen¹

¹ Pervasive Computing, Tampere University of Technology, Tampere, Finland,

² City of Tampere, Tampere,

³ Aalto University, Espoo, Finland and Linköping University, Linköping, Sweden,

⁴ Computer Science, University of Tampere

Abstract. Computing is the latest add-on to enhance the K-12 curricula of many countries, with the purpose of closing the digital skills gap. The revised Finnish Curriculum 2014 integrates computing mainly into math. Consequently, Finland needs to train math teachers to teach computing at elementary level. This study describes the Python and Racket tracks of the Code ABC MOOC that introduce programming basics for math teachers. Their suitability for math is compared based on the course content and feedback. The results show that conceptually the functional paradigm of Racket approaches math more closely, in particular algebra. In addition, Racket is generally regarded as more challenging in terms of syntax and e.g. for utilizing recursion as an iteration mechanism. Math teachers also rank its suitability higher because the content and exercises of the track are specifically tailored for their subject.

Keywords: Curriculum Research, Computer Science Education, K-12 Education, In-Service Teacher Training, MOOC, Computational Thinking, Math-integrated Computer Science, Python, Racket, Programming Paradigms, Imperative, Functional.

1 INTRODUCTION

Our society is becoming increasingly digitalized, which has therefore given rise to a global discussion on the role of computer science (CS) in K-12 education. As a consequence, a number of countries all over the world have introduced computational thinking, computing or CS (or aspects thereof) into their K-12 curricula. Since 2014, for instance, students in England have had Computing on their schedule from the age of five. In Finland, aspects of CS were included in the national curriculum in fall 2016, when the 2014 national curriculum came into force. Programming was introduced as part of the cross-curricular theme of digital competence, and also specifically integrated into the syllabi of crafts (Y3-9) and mathematics (Y1-9). In Y1-2, math teachers now need to help students learn how to create and test simple programs (unplugged, step-by-step instructions), while students in Y3-6 should learn how to program in a visual programming language. The new learning objectives for mathematics in Y7-9 intend to develop students' algorithmic thinking skills and applying programming in problem solving. The target is reached when "*a student can apply the principles of algorithmic thinking and is capable of implementing simple programs*" [15].

Integrating programming into comprehensive education is a remarkable change: both pre- and in-service teachers need to learn to program and to understand the core elements of computational thinking. Such curriculum enhancements change the job description of a teacher retrospectively. Consequently, employers are responsible for the need to train teachers and for providing time for sufficient professional development. Although this training need is recognized by the government, in-service training resources are insufficient as Finnish teacher training departments have not yet fully responded to the new requirements and the growing need.

Against this background, all voluntary training initiatives have been warmly welcomed by schools, principals and teachers. In this paper, we present the Code ABC MOOC, a project initiated informally by a group of volunteer educators to respond to the gap in teachers' formal training and preparedness for teaching programming. The initiative was later sponsored by the Technology Industries of Finland Centennial Foundation and the Finnish National Board of Education. The Code ABC MOOC offers four tracks targeted at teachers working at different school levels: ScratchJr (Y1-2), Scratch (Y3-6), Racket (Y7-9) and Python (Y7-9). In addition to supporting elementary school teachers in learning the basics of programming, the Code ABC MOOC also serves as a tool for highlighting best practices for teaching programming. This paper concentrates on the two tracks targeted at teachers of Y7-9, namely the Python and Racket tracks.

In this study we extract the key concepts and aspects of programming and computational thinking from the examined tracks of the Code ABC MOOC in an effort to strengthen the conceptual basis of the programming syllabus. The teacher feedback illustrates how these concepts have been adopted and evaluates the suitability of the material for supporting teachers in adopting the new curriculum. In addition, our study attempts to link these CS concepts to appropriate mathematics topics. In this regard, the differences between the programming languages used in the two tracks are noted and explained based on the underlying programming paradigms. Our goal is to answer three main questions:

- What CS concepts and computational thinking skills do these Code ABC tracks introduce?
- What topics did the teachers find challenging, inspiring, or suitable for math?
- Which programming paradigms do the tracks align with and how do they support math?

The paper is organized as follows. First we discuss previous work in fields related to our study, after which we describe our research context. Next we present and discuss the results, before concluding the paper with some final remarks.

2 RELATED WORK

2.1 CS in K-9 education

As noted in the introduction, introducing aspects of CS in K-9 education is an international trend. This is, however, not a new trend. As long ago as 1967, Papert developed the LOGO language [35], specifically aimed at teaching children how to program. His goal was to use programming as a tool to let children be creative with technology and to support their learning in other domains such as mathematics, the arts, languages and science. As computers became increasingly popular, accessible and easy to use, the focus in the school debate shifted from CS and learning how to program to IT and learning how to use computers and software. In the last five years, the trend has once again shifted, as the digital transformation has shown the need for understanding the digital world. Consequently, there has been an active debate on the need to shift the focus away from our future citizens being mere passive consumers of technology, and towards them becoming active producers.

Internationally, we now see an increasing trend for intergrating aspects of CS into K-9 education. For instance, in Europe the majority (17 out of 21) of countries participating in a survey conducted by the European Schoolnet in 2015 reported doing so [4]. The way in which this is accomplished varies. Some countries focus on K-12 as a whole, whereas others primarily address

either K-9 or grades 10-12. Some countries have introduced CS as a subject on its own (e.g. Computing in England) while others have decided to integrate it with other subjects, by, for instance, making programming a cross-curricular element (such as in Finland). Instead of computing and programming, some countries also use the term computational thinking.

Computational thinking (CT) has gained prominence, particularly in conjunction with the discussion on integrating aspects of CS as part of K-9 education. However, Papert had already set the course for CT much earlier, in 1996 [34]:

“Computer science develops students’ computational and critical thinking skills and shows them how to create, not simply use, new technologies. This fundamental knowledge is needed to prepare students for the 21st century, regardless of their ultimate field of study or occupation”. Not simply using computers but also creating digital artifacts is a valid stance – helping pupils to identify themselves as potential creators fosters their sense of empowerment.

In her seminal article, Wing [58] renewed this emphasis by establishing the term ‘computational thinking’ as an essential part of the recent CS education discourse, yet its comprehensive definition was omitted. To date, several definitions and operational descriptions of CT have been suggested. Even if no consensus on the definition has been reached, many of the suggestions build on the core of Wing’s later description (2010, [59]): *The thought processes involved in formulating problems and their solutions so that the solutions are represented in a form that can be carried out by an information-processing agent.*

When operationalizing the term, it is commonly presented as a set of concepts and approaches. For instance, the support organization Computing at School (CAS) in England, describes CT as “the processes and approaches we draw on when thinking about problems or systems in such a way that a computer can help us with these.” (CAS, p. 1) They define CT in terms of six concepts (logic, algorithms, decomposition, patterns, abstraction and evaluation) and five approaches (tinkering, creating, debugging, persevering and collaborating).

Relevance for our study In the Finnish curriculum programming is to be taught as part of math [15]. Compared to programming, math already has a well-established syllabus that has evolved into its current state since the dawn of the educational system. Apart from a few minor syllabus areas being dropped from or reintroduced into the curriculum, the core content of the math syllabus has remained more or less the same for decades. In order to facilitate a smooth transition, there is a need to build on the well-established math core in order to introduce the analogous and logically progressive steps for CS. It is tentatively assumed that integrating CS into math will move the center of gravity of the syllabus towards CT.

2.2 Bidirectional transfer: *math* ↔ *CS*

Math is at the very core of programming, which requires algebraic, logic and problem-solving skills. Synergy implies mutual benefit between two entities, and although the benefits that a good understanding of math and perceived self-efficacy confer on the learning of computational skills are clear [62,44], the transfer in the other direction, from programming to math, may not be that obvious. In a successful transfer, however, a student should be capable of finding the common underlying conceptual bases of different topics [23]. Finding such analogies not only requires a certain level of intellectual maturity, but also that a student has elaborated on the learning material conceptually in order to reach a deeper understanding of the topic.

Transfer may happen either laterally or vertically [17], near or far, or by the low road or the high road [40] implying a certain hierarchy of learning. In addition, one of the two complementary subjects tends to be interpreted in the learners' minds in a more abstract manner, while the other focuses on its application [43]. In the case of math and CS, math is more abstract, while CS can be understood as applied math [8]. In math, educators have long talked about conceptual and procedural knowledge [18]: conceptual knowledge comprises a full possession of the appropriate concepts and the ability to link them together, i.e., the high road to knowledge transfer, while procedural knowledge (the low road) consists of well-internalized mathematical routines.

Relevance for our study Based on previous research, one potential assumption is that practicing math routines will provide an appropriate resource for programming exercises. To achieve this, the current math syllabus needs to be bridged with the corresponding programming topics. It seems reasonable to assume that this could be valuable not only to students, but also for in-service teachers, who might find similarities between math and programming motivating when learning to program themselves.

2.3 Programming metaphors, languages and paradigms

When attempting to determine the role of CS in education, various metaphors are used, e.g. thinking of programming as literacy, as a driver for the maker culture and a maker mind-set, or grounded math [7]. If the literacy metaphor is used, then programming as digital literacy emphasizes the same logical skills as are applied when constructing linguistically correct sentences, such as using and/or/not in order to express the internal logic of a sentence. From a 'maker mindset', the programming language should be as productive as possible, easy to learn ("low floor"), usable in a wide range of potential application areas and types ("wide walls") and facilitate the creation of both basic and advanced programs ("high ceiling"). In such contexts, scripting languages and visual programming languages such as Scratch (scratch.mit.edu) can be particularly useful.

The question of what programming language to use has been heavily debated throughout the years, in particular when discussing novice programming. At university level, most discussion has surrounded textual languages, such as Java, C, Python and Scala. Languages have been compared based on, for instance, how easy they are to learn, how many errors students make when using them, how verbal the languages are and potential syntactical overload. However, at the K-9 level this question has not been as actively discussed, for a number of reasons. First, programming at school level is still rather new, and thus still evolving into its final form. The goal of learning the basics of programming at school is not to educate future programmers, but rather to give them a basic understanding of the digital world. In addition, at the moment, educators seemed to reach a consensus about the programming progression at school, starting with unplugged activities, followed by visual programming languages with a textual language being introduced in the later grades (7–9). Nevertheless, there are some studies on programming languages at K-9 level as well. Despite the popularity of Scratch and other block-based languages, some studies have questioned the benefits of these in enhancing problem-solving skills and good programming practices [22,33] as these are claimed to lead to bottom-up development and fine-grained programming without coherence [33]. In sum, the problems relate to abstraction skills, i.e., "not seeing the forest for the trees" and designing the program in advance. On the other hand, other studies have found that visual programming languages such as Scratch make it easier for novices to learn some concepts, for instance, construct of conditions [27,31] and to switch to textual programming later on. Another consequence of using

block-based languages is the tight integration with the development environment (IDE), whether on-line or stand-alone. Such powerful IDEs become a new norm along with visual programming.

In addition to metaphors and languages, programming paradigms are essential for defining the angle of approach to teaching programming. Each paradigm provides its own basic concepts with paradigm-specific restrictions, which leads to different kinds of implementations and programming styles. Some tasks are more easily implemented in one paradigm, whereas other paradigms are more appropriate for other applications (e.g. due to their efficiency or flexibility). Consequently, there is not only a discussion around what programming language to use, but there are also recurring arguments about “the right paradigm for the job”. To be able to make well-informed language and paradigm choices, decision-makers and educators should have an adequate understanding of the alternatives available.

The division of programming languages into different paradigms is not easy, and is further blurred by multi-paradigm languages. Wegner (1989) simply divided languages into two fundamental categories: imperative and declarative [55]. In this division, the imperative paradigm comprises procedural, object-oriented, and distributed (parallel) languages, whereas the declarative one consists of functional, logic, and database languages. However, a parallel paradigm is not commensurate with, for instance, an object-oriented paradigm, which provides for parallel implementations as well. This problem can be circumvented by separating the programming model (sequential/parallel) from paradigms altogether, thus enabling new combinations. This change was proposed by Bal and Grune [3], who also raised the previous sub-paradigms of procedural, object-oriented, functional and logic to main categories.

Constantly increasing in number, multi-paradigm languages challenge this categorization. For instance, Scratch puts paradigm categorization to the test. Some of the features of Scratch comply with an object-oriented paradigm. According to Van Roy’s taxonomy, having a cell and a thread, i.e. assignment and concurrency, categorizes Scratch as an object-oriented paradigm [54]. However, the lack of data abstraction (inheritance) and functions makes the data encapsulation model of Scratch oxymoronic in regard to object-orientedness. Since Scratch targets only GUI applications reactive to user-generated events and inter-sprite messages, an event-loop [54] or event-driven paradigm [14] would seem to be a more accurate categorization. A few sources refer to Scratch as agent-based programming, where each sprite acts as an independent agent according to the defined rules. The interplay of such agents, for instance, facilitates easy STEM simulations [51].

Relevance for our study Since our study compares two tracks of the Code ABC MOOC, one using a functional language and the other an imperative one, this discussion of paradigms is important. In this section we look more closely at the paradigms most relevant to our study: imperative and functional programming, exemplified by Python and Racket respectively.

Imperative paradigm and Python Some argue that the imperative paradigm is appropriate for introducing programming as it makes it quite straightforward to translate algorithms into code, for instance. There are a wide range of imperative languages which can be used as general purpose languages and for particular application areas. Python (python.org) was originally designed with education in mind. The developer, Guido van Rossum, even suggested that everybody could master programming using Python back in 1999 [45]. The language had already aroused interest as a programming language for novices in the early 2000s, due to, for instance, its clear and readable syntax, strong introspection capabilities, natural expression of procedural code, high level dynamic

data types and its extensive standard library and third party modules providing functionality for a wide range of tasks. Python is one of the most commonly used languages in general use today (number 5 on the TIOBE programming index list in August 2017) and is widely used in education [21,10,9, etc.].

Guzdial [21] has a clear vision of the importance of web programming and sees Python as a viable tool for this, describing it as "one of the best web languages". However, he does not object to mixing paradigms and languages, exemplified by the Jython environment for students (JES) project, which combines Python with Java [5]. Nevertheless, Python is object-oriented even without Java by virtue of its own class model that provides e.g. polymorphism and multiple inheritance. Python easily interweaves multiple paradigms, although at the basic level it does fall into the imperative paradigm. Recently, Python was endowed with functional features as well, such as maps, comprehensions, and generators [28].

The Python material of the Code ABC track has been translated and modified based on the project outcomes of AP Computer Science Principles [19]. Already in 2003, Guzdial [20] championed the 'CS for all' ideology and the potential of CS over specific syllabus areas of math, such as calculus. He argues that teaching CS for all – not just for mathematically oriented students – should involve "sampling" instead of sorting. It is essential to allow space for students' authentic interests, such as arts, crafts, and music in the hunt for intrinsic motivation. Incorporating the maker mindset with tinkering, and creative and socially meaningful activities is especially beneficial for reaching less motivated student groups, including girls [6].

Functional paradigm and Racket In contrast to the multi-faceted nature of Python, the subset of Racket used in the Code ABC MOOC is more constrained and the closest metaphor would be "grounded math", where the pure functional programming language may be regarded as a realization of lambda calculus. Transfer between math and CS is claimed to be closest to the functional programming paradigm [26,49]. For example, functions in algebra can be practiced using functional programming languages. Combining functional programming with math is not new. Historically, attempts range from the early use of LOGO [16,25] to recent experiments employing Racket and Haskell [1]. While the results from the LOGO initiatives have varied [25], the Racket evaluations have been positive and stable [13,12,49,47].

The Racket programming language (<http://racket-lang.org>) is a multi-paradigm language, and thus also supports functional programming. Racket includes a programming IDE, DrRacket, designed especially for teaching purposes [13]. In contexts where DrRacket cannot be installed, the web-based environment WeScheme [61] steps into the breach, also enabling online sharing and remixes. DrRacket has built-in support for so-called 'Student Languages' starting with Beginning Student and ending with Advanced Student Language. Each of these Student Languages gradually introduces new programming primitives and concepts. Simplified syntax and semantics aim at helping beginners grasp the core concepts of function design, such as composition and function calls. Tool creators have also defined more precise error messages in order to assist novices in debugging and analyzing their code [30].

For the sake of the purity of the functional paradigm, the imperative features of Racket are held back. For instance, the assignment operation (set!) and functions causing side effects (display, read) are not introduced until the Advanced Student Language level. Felleisen et al. wrote the guide 'How to Design Programs' for high school and college level students [12], and in its most recent version the imperative features are done away with altogether to introduce appropriate coding practices.

The guide systematizes problem solving with Design Recipe, which teaches how to divide a problem into smaller solvable steps in the process of designing functions with a test-driven approach [12]. The use of Design Recipe has been shown to foster the right order of operations and the composition of nested functions. Thus, Felleisen and Krishnamurthi suggest that functional programming provides the strongest evidence for the favorable effects of programming on math skills [13].

A number of articles promote Racket’s Beginning Student Language as a prominent way of learning algebra [26,49], especially with well-designed instructions. These should include purposefully planned exercises and pedagogical models, such as the Cycle of Evaluation [49], which visualizes expressions and the use of parentheses. The algebraic approach of functional programming has been shown to improve the understanding of math concepts such as variables and functions [60,49,48].

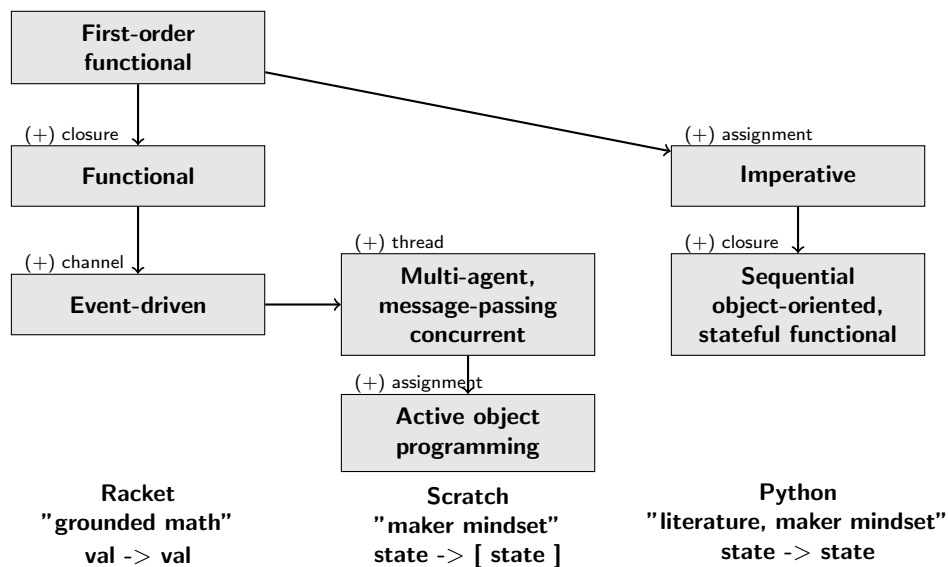


Fig. 1. The Code ABC paradigm taxonomy complying with [54]

Comparison and Summary of the Paradigms of the Code ABC In summary, Figure 1 illustrates the Code ABC, which is comprised of functional, event-driven, and imperative paradigms. In line with the paradigms, the figure also illustrates the corresponding language used in the MOOC and the closest metaphors. Obviously, these metaphors are speculative. Different "paradigm camps" tend to adhere to their own discourse: Scratch promoters, led by Resnick, highlight sharing and the unimpeded creation of one’s own artifacts with the analogy of virtual LEGO construction, i.e. block snapping [42,32]. The founder of the Python language, Van Rossum, emphasizes the readability and efficiency of code [46], whereas the Racket camp regards functional computing as being rooted in lambda calculus, thus inherently connected with math, in particular algebra [49,48,13].

Van Roy categorizes languages based on their declarativeness and expressiveness [54,53]. In his fine-grained paradigm taxonomy, Van Roy defines a horizontal axis of declarativeness based on whether a state is unnamed or named, and adds expressiveness step-by-step (for instance, assignment, closure, channel, and thread) in order to evolve the paradigm taxonomy from simple to more

complex concepts. In distinguishing between functional and imperative paradigms, the diagnostic question is: can you assign a variable, i.e. have a named state? The answer divides paradigms into either imperative or functional. An imperative paradigm is statement-centric, the assignment being a statement as well. Each statement changes the state of the system, hence, imperative computation may be understood as state transformations in a sequence, i.e., $state \rightarrow state$. This is in contrast to the functional paradigm, which may be described as sequential value transformations, $val \rightarrow val$ without states. A named state enables modularity and the storage and management of updatable memory, which moves the paradigm in a less declarative direction.

A closure establishes a new variable scope in the context of a function. It 'closes' both a pointer to code and an environment for free variables. In the functional paradigm, closures are a central concept because they enable nested and higher-order functions that can access data from the outer scope, i.e. variables of the previous frames in the stack. Higher-order functions are a powerful substitute for e.g. for/while iterations without incrementable counters. Otherwise, control structures would be cumbersome to implement.

The event-driven paradigm leans on events that trigger execution, e.g. at the user's initiative. An event may trigger a message to be broadcast through a channel (or a port). In Scratch, several loosely-coupled receivers may listen to the same message [29]. Because the order of receivers is not determined, the concurrency model comprises explicit sending and implicit receiving, which implies a non-deterministic final state, i.e., $state \rightarrow [state]$. Infinite forever loops are implemented as threads that enable concurrency.

3 The Code ABC

The initial idea for the Code ABC MOOC was introduced in spring 2015 by Tarmo Toikkanen and Tero Toivanen. The original concept was to help teachers learn programming using material that has been prepared especially for them by their peers, for instance, more experienced teachers. The first course, the so called Code ABC beta, was held in Autumn 2015 with three tracks: ScratchJr, Scratch and Racket. The Python track was added for the spring 2016 version of the MOOC. So far (fall 2017) 3649 participants have studied programming in the Code ABC MOOCs.

The initial three tracks of the Code ABC beta (ScratchJr, Scratch, and Racket) were developed by a group of Finnish

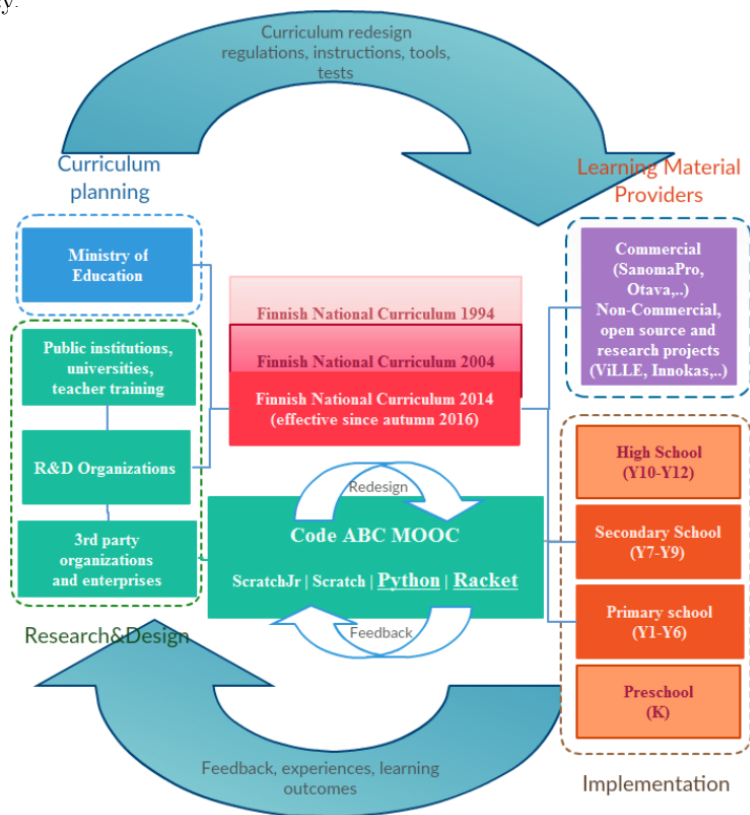


Fig. 2. Nested DBR cycles of curriculum updates (update/10yrs) and Code ABC tracks (2 updates/yr)

teachers and were improved incrementally based on the feedback from several iterations. The continuous development followed the principles of design based research (DBR), aiming at linking theory and practice in the discipline of education [41]. DBR stipulates the use of several iterations and redesigns of an educational artifact based on feedback and experience. Figure 2 illustrates the process of two nested design cycles.

The outer cycle is the curriculum planning cycle that takes place once a decade, while the inner is the iterative process of developing the Code ABC tracks twice a year. Development proceeds in cycles, taking into account the feedback given by different stakeholders, especially the customers, which in this context are in-service teachers. The artifact is then redesigned together with course instructors and researchers, whose research interests lie in integrating CT into elementary education.

The original material for the Python track was developed in the USA in similar cycles by Guzdial and Ericson [11]. This material was translated and adapted for the Code ABC MOOC in a project funded by the Finnish National Board of Education, coordinated by the Innokas Network at the Faculty of Educational Sciences at the University of Helsinki and implemented jointly by the Faculty of Educational Sciences and the Department of Computer Science at the University of Helsinki and the Department of Computer Science at Aalto University.

3.1 Design goals of the Code ABC tracks

The first three tracks of the Code ABC (ScratchJr, Scratch, and Racket) had a number of general goals: promoting creativity; introducing CS as a tool for creating something new and inspiring; sharing pedagogical ideas and artifacts during the course; using exercises directly applicable in a classroom context in order to make it easier for teachers to get started; offering course participants sufficient content knowledge so that they would not limit themselves to applying ready-made programming materials but also be able to create their own exercises; and enabling peer-support by encouraging participants to help each other on discussion forums. The Racket track had an additional design goal of integrating mathematics into programming exercises in order to motivate math teachers to adopt programming in their teaching, and to prove that programming lessons are not time wasted.

The main goals of the Python track were, according to the goals of the original material [10], to increase teachers' knowledge of computer science concepts as well as to improve teachers' confidence in their ability to teach CS principles. The course was designed as a lightweight learning experience [11], allowing busy teachers to participate when they had 20-60 minutes to spare. The exercises were designed to be small and feature low cognitive loads, which was achieved by placing relevant examples just before the exercises. The course did not focus on any individual subject such as mathematics – on the contrary, the material was aimed at anyone interested in teaching programming and CS.

3.2 Course implementation

The Code ABC MOOC was implemented using the A+ learning platform developed by Aalto University (<https://plus.cs.hut.fi/>). Piazza was utilized as the discussion platform and Rubyrice for showcasing and peer reviewing returned artifacts [2]. Both the Python and the Racket tracks grouped learning objects into entities; termed modules in Python, and topics in Racket. For the sake of consistency, we will use 'topic' for both in this paper. At the end of each topic, feedback was collected with Grader, an on-line survey tool developed at Aalto University. Grader was also

used to collect pre- and post-course surveys. The feedback was collected in order to further develop the courses. Teachers attended the Code ABC MOOC tracks free of charge. Both tracks spanned several months from February to May, 2016. No compensation was granted for course participants except for 2-3 credit points from the Open University of Helsinki after course completion (2 cp for Python (P), 3 cp for Racket (R)).

4 Study design and data collection

Our study is based on the Spring 2016 course implementation consisting of the second iteration of the Racket track and the Finnish translation of version 2 of the Python material. We conducted a pre-course survey to get background information about the participants (Python N=320, Racket N=137).

The largest participant group represented 25 to 35 year-olds (Fig. 3) the majority of whom were female (Python track 65%, Racket track 78%). Math teachers formed the largest groups (Fig. 4) in both tracks (Python track 48%, Racket track 88%). The next largest groups were science and CS teachers. The majority of the participants represented the original target group, i.e., K-9 teachers (Python track 72%, Racket track 91%).

Based on the survey, most participants had some previous experience in programming (Python track 78%, Racket track 74%). Quite a few had already used programming in their teaching (Python track 48%, Racket track 38%). In order of popularity, the languages previously used by the Python track participants were Scratch 34%, Java 32%, Basic 28%, Pascal 25%, C++ 23%, JavaScript 21%, C 18%, Python 17%, Visual Basic 17%, FORTRAN 12%, LOGO 11%, C# 6% and Perl 5%. In Racket, the order of popularity was Scratch, 34%, Java 26%, C++ 23%, Pascal 22%, Basic 20%, Python 15%, Visual Basic 14%, JavaScript 10%, FORTRAN 9%, LOGO 8%, and C 7%. Other languages were only mentioned by 5% of the participants or less.

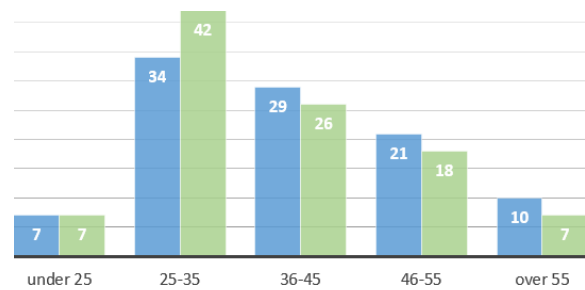


Fig. 3. Age distribution of course participants in % (P blue, R green)

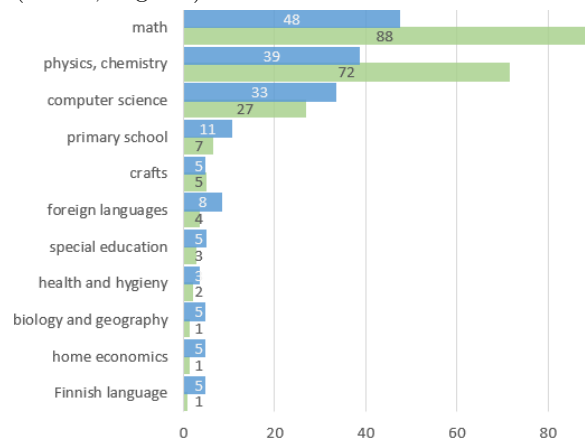


Fig. 4. Subjects taught by the teachers (P blue, R green; a subject omitted if P and R less than 5%)

4.1 Python and Racket tracks in Spring 2016

The Python track (content described in Table 1) was implemented as a localized translation of the ebook Computer Science Principles: Big Ideas in Programming [10]. The course was organized in five topics in accordance with the original material: introduction to computing, naming, repetition, decision making, and data manipulation. Participants concluded the course by writing an essay on the pedagogical aspects of programming.

The original material is arranged in a book format, and should thus be read in a sequential manner. All the exercises are embedded in the ebook’s browser environment and they are automatically assessed and graded. The material has been designed to follow an example-exercise format to facilitate learning [10]. Multiple exercise types are used: multiple choice and fill-in-the-blanks test conceptual understanding, Parson’s problems are used for teaching basic programming constructs, and exercises consisting of modifying active code segments in the on-line programming environment provide wider opportunities to try out the concepts learned [36]. In addition, the material utilizes the code lens concept to demonstrate program execution [11].

As the original material did not specifically focus on how programming should be taught in Finnish schools, the Code ABC MOOC had an additional course project, during which the teachers designed a 2-hour lesson on any subject of their choice. The course projects were reviewed by both peers and course staff. The lessons that the teachers had designed typically reflected the subjects that they taught in schools, ranging from learning languages to applying CS in crafts. The participants had to complete 85% of the automated exercises and the final essay in order to pass the course.

The Racket track was designed so that different aspects of algorithmic thinking (abstraction, logic, repetition) were introduced side by side, starting from easier ideas and progressing to more advanced topics [39]. Altogether, the course content comprised seven topics: introduction to programming, control structures, functions and design recipe for functions, recursion, user interactions, lists, and higher-order functions with Turtle graphics (content described in Table 1). The very core of the Code ABC Racket track material is to reveal the nature of programming as a sort of applied mathematics, and to show how mathematics can be taught through programming. Hence most programming exercises are in a math context. The implementation of the Racket track was inspired by the Systematic Program Design online course offered by edx.org [24] and the material was constructed following the same procedure:

1. Short motivational video, in which the lecturer introduced the contents and the purpose of the exercises. A few videos also responded to questions from the previous week.
2. Tutorial videos introduced the core concepts as screen captures. The lecturer demonstrated the concepts to be learned with DrRacket. Its stepper tool was extensively employed in explaining the evaluation rules. Concise lecture notes were available on-line as well, but the majority of the course content was provided as videos. The course participants were expected to test the programming examples themselves while watching the videos.
3. The Design Recipe was used to demonstrate the principles of function design, see Figure 5. By using the recipe, a user can solve one detail at a time and proceed step-by-step until the whole

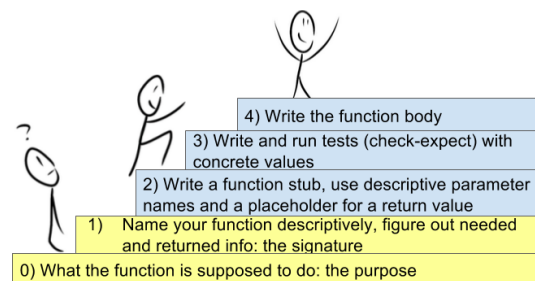


Fig. 5. The Design Recipe [12] presented as a staircase fostering a step-by-step design [37]

function is ready. The implicit intent is to solidify writing test cases before implementing the actual function body, which complies with the test-driven development.

4. The exercises and their solutions were delivered as both DrRacket and WeScheme files and used as self-tests of the course content presented in the video tutorials.
5. Hands-on exercises differed from the Systematic Program Design exercises as self-review for code and multiple choice quizzes were not used. Also, the final course project was an essay about the pedagogical aspects instead of a programming project as in Systematic Program Design.

The programming exercises and their solutions were taken from the Coding for schools student’s material [38] and the Coder’s handbook [37], which contains documentation for the graphics and animation libraries (2htdp/image and 2htdp/universe), Beginning Student Language primitives, and new libraries for turtle graphics (Racket Turtle) and user interactions (display-read).

Participation and course completion levels Fig. 6 shows the number of participants per topic. By ”enrolled” we mean participants who registered to the MOOC, whereas ”started” refers to participants who showed some activity in the first topic (completed one exercise or gave feedback). The numbers 1-7 refer in Racket case to participants who completed the corresponding topic. The numbers 1-5 in Python case refer to participants who completed the feedback form at the of each topic. The Racket track was started by fewer participants (N=171) than the newly introduced Python track (N=399). Both courses lost participants during the course period, but in the end more participants completed the Racket track (N=100, 58% of the started participants) than the Python track (N=66, 17% of the started participants). If the percentages are calculated per enrolled participants we get lower values for completion rates: Python 12% and Racket 31%. Only 80% of the course work was required to pass in Racket track, which allowed skipping one topic. This explains the lower numbers for topics 5, 6 or 7 in Fig. 6.

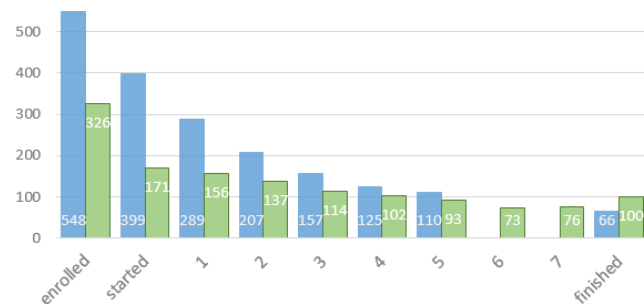


Fig. 6. Number of participants/topic (P blue, R green)

4.2 Research methods

The methods of the study are curriculum research and content analysis of the participants’ feedback. Curriculum research examines the most central concepts of the curriculum, which in our context correspond to CS concepts in the Code ABC MOOC. In deriving the main concepts, we counted their frequency in content, that is, how often they occurred. We also asked participants what they had learned after each topic, and their responses were similarly analyzed by counting the occurrence frequencies. After extracting the most central CS concepts, the paradigm-oriented differences were examined more thoroughly. The goal of the review was to check how good a match the respective paradigms were for teaching math.

5 Results

5.1 What CS concepts and CT skills do the Python and Racket tracks introduce?

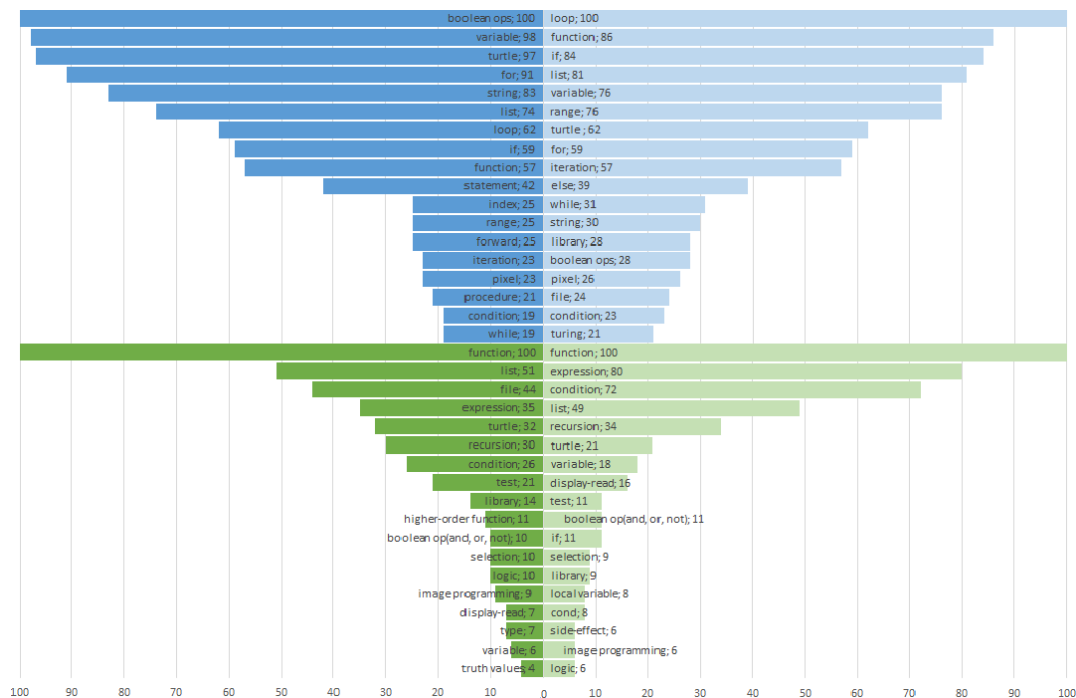


Fig. 7. To the left the relative word frequencies of the MOOC material (P blue, R green); to the right teachers' feedback to the question "What did I learn?"

Figure 7 illustrates the topics taught vs. the topics learned for each course. The brighter-colored bars to the left correspond with the relative word frequencies in the course content. The most frequent word has a value of 100%, and other frequencies are compared to this maximum. The lighter-colored bars to the right represent the course content based on participants' responses to the question "What did I learn?".

In the figure, the blue Python tornado lies above the green Racket tornado. From the shape of the tornados we can conclude that the wider Python tornado covers a larger range of concepts with about equal intensity. In contrast, the Racket course focused on functions and a handful of other concepts. The relative similarity of the mirrored right side indicates that the participants in both tracks seemed to learn the intended concepts.

The Python topics in the upper blue tornado indicate that the main concepts of the Python track were control structures (selection and iteration). Boolean operators (and, or, not) were widely exploited in the program examples, including conditions for iterations, such as for and while loops. All selection and iteration-related topics appear in the list: boolean ops (1st), for (4th), loop (7th), if (8th), index (11th), range (12th), iteration (14th), condition (17th), while (18th).

Naming and variables (2nd) were the second most central concepts, reflecting the stateful and assignment-oriented nature of Python as an imperative language. We also group statements (10th) here, as a superclass, including functions (9th), procedures (16th) and assignments. By comparing

functions and procedures the material highlighted the difference of functions returning a value and procedures lacking return values.

The third topic group were applications, exemplified by turtle (3rd) and pixel-level image editing (15th). Concept-wise, these applications do not bring anything new, but rather give students the opportunity to put the pieces together while working on engaging problems. The last group consists of data types and structures of string (5th) and list (6th); in the course, strings are named (used as variables) and lists are iterated. So, list would fit in the iteration of the first group as well.

The participants echoed these emphasis areas, apart from the Turing machine and file areas, which received more emphasis than they had in the text. The Turing machine, completeness and halting problems, were used to explain the history and most prominent ideas of CS. Files were introduced in the last topic (Data handling) simultaneously with related functions such as `open` and `close`. Statements and `index` were not mentioned.

The analysis of the Racket course concepts is only partial for technical limitations; we were not able to analyze video material since it was not transcribed into textual format. The analysis is based on the lecture notes and textual material in the course platform, so the analysis might be missing some concepts that were taught but are not showing in Fig 7.

In the green Racket tornado functions, expressions, conditions, lists, recursion and Turtles form the top 6 concepts mentioned on both taught and learned sides. Testing, boolean operators and library usage are mentioned next. The high frequency of the 'file' concept on the lecture side is explained by the fact that the term appeared frequently throughout the course instructions ("Save your code in a file", "Send your file for peer-review", etc) even though file handling was not explicitly taught. The participants reflected, quite faithfully, the same concepts except higher-order functions and type, which were missing from the list of concepts mentioned by the participants.

5.2 What topics did the teachers find challenging, inspiring, or suitable for math?

The participants evaluated each topic after completion using 5-point Likert-scales. The evaluation was based on a set of criteria such as difficulty level, enthusiasm, and suitability for their teaching. The following subsections present the results of these evaluations.

Challenging topics The average level of difficulty experienced for each topic is given in Fig. 8 (1=not challenging enough, 5=far too challenging). For both tracks, the difficulty level increased during the first four topics. After the fourth topic in the Racket track, the difficulty level decreased, whereas it continued to increase in the Python track. Topic 4 in the Racket track (recursion) was considered most challenging. Animations (topic 3) and lists (topic 5) are the next most challenging. Starting from topic 5 the challenge level in Racket starts

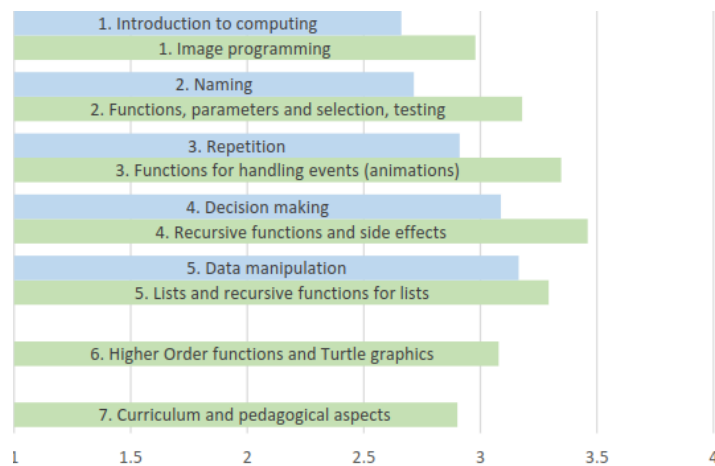
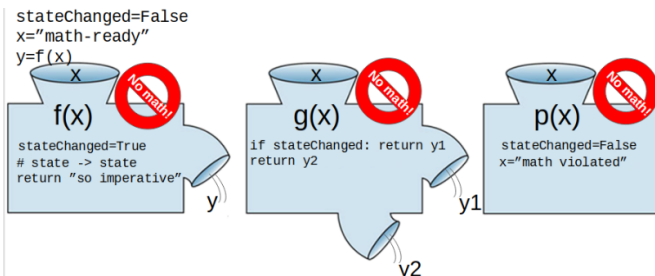
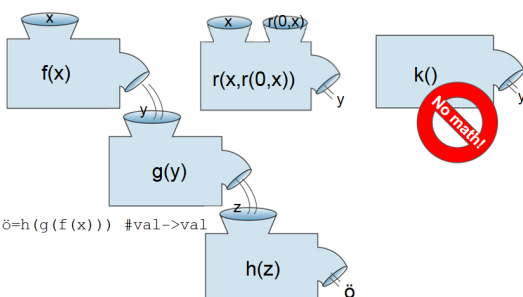
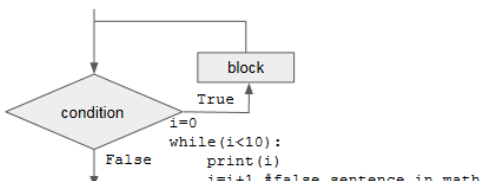
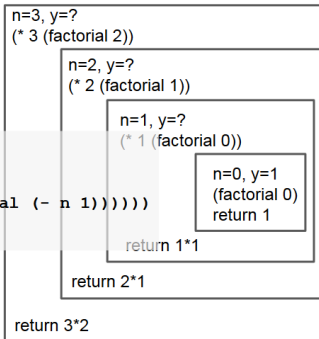


Fig. 8. Difficulty level/topic (P blue, R green)

Table 1. Fundamental concepts and CT related aspects of the Python and Racket tracks with regard to the underlying paradigm.

Python (imperative)	Racket (functional)
<p>1 Basic operations, computing. What is a computer, program (Python vs. Java)? Background, Turing machine (completeness).</p> <p>2 Naming (including variables) applied to numbers, strings, objects such as turtles and images, as well as functions string: substrings, indexing. Functions ($f(x)$, $g(x)$) vs. procedures ($p(x)$).</p> <pre>stateChanged=False x="math-ready" y=f(x)</pre> 	<p>Introduction to Racket programming, global constants. CT: problem decomposition using functions</p> <p>Control structures: selection (<code>if</code>), truth values, comparisons. Function definition: purpose, signature. Unit tests using <code>check-expect</code></p>  <p>$\delta = h(g(f(x)))$ #val->val</p> <p>*)The $k()$ is introduced in the topic 4.</p>
<p>3 Control structures: iteration (<code>for</code> and <code>while</code>). Iteration based on list or counter condition, e.g., a list [1,2,3] iterated based on range: <code>for item in list: block</code> <code>while (condition): block</code> Block: indentation-grouped commands CT: becoming aware of iterative patterns.</p>	<p>The Design Recipe[12]: test-driven development. Selection (<code>cond</code>): comparisons, predicates. Logical ops for combinations: <code>and</code>, <code>or</code>, <code>not</code>. Interactive applications: animations, mouse events CT-abstraction: Design Recipe.</p>
<p>4 Blocks by indentation. Control structures, also nested, selection: <code>if</code>, <code>elif</code>, and <code>else</code>. Decision making: – condition (logical expression) in iteration or selection – logical operators for combinations. <code>if condition: block</code> <code>else: print("condition false")</code></p> <p>CT abstraction: flowchart illustrating the control flow.</p> 	<p>Reading user input with <code>display-read</code> (user interaction causes side-effects); the user input stored into a local variable. Recursion, here factorial $n!$ as an example:</p> <pre>(define factorial (lambda (n) (if (= n 0) 1 (* n (factorial (- n 1)))))) (factorial 3) 6</pre> 
<p>5 Data handling: collections and files. Collection operations:</p> <ol style="list-style-type: none"> 1. indexing 2. string: <code>split</code>, <code>find</code>, <code>substring</code> 3. list: <code>len</code>, <code>range</code>, <code>for-each</code>. <p>Reading files: <code>open</code>, <code>close</code>, <code>readlines</code>. Conventions: commenting.</p>	<p>Iterating lists recursively, producing new lists or one accumulated value. Image files.</p>
<p>6 Revision, extra material.</p>	<p>Iterations cont. Higher-order functions: <code>map</code> and <code>apply</code>, lists.</p>
<p>7 Finnish curriculum reflections in regard to CT/CS, and how to integrate computing with own teaching subject.</p>	<p></p>

to decrease. Explanation might be that there were less exercises in topic 5 than in the previous topics. Also topic 6 covered Turtle graphics programming, which is conceptually simpler than functions and recursion.

The list below provides a few free-text feedback snippets describing challenging topics:

Python

- Repetition: a number of participants were not capable of constructing loops on their own
- Decision: the difficulty level rose sharply compared to the previous modules; too much information and challenge
- Image processing: exercises felt hard and difficult to understand for beginners, and did not foster learning repetition
- Data manipulation: A few exercises were too difficult, e.g. in searching data from a file, one really needs to know what each function returns

Racket

- The most challenging topics included recursion, animation, lists, and loops
- Recursion: content of topic 4 was clearly too much. It should have been split into two separate topics
- Lists: topic 5 also challenged a number of participants

Inspiring topics The average enthusiasm score for each topic is shown in Fig. 9 (1 = not inspiring at all, 5 = extremely inspiring). The highest levels of enthusiasm were reported in Racket for Turtle graphics (topic 6) and Image Programming (topic 1). This is in line with Toikkanen’s [52] findings of the mesmerizing effect of Turtle throughout all the Code ABC MOOC tracks. Young students immediately start drawing Logo-like figures after discovering the `pen.down` function. In addition, animation (topic 3) inspired a good number of participants.

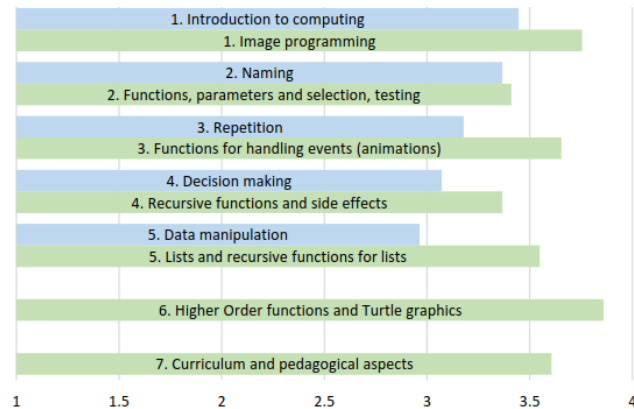


Fig. 9. Level of enthusiasm/topic (P blue, R green)

For the Python track, the difficulty level (Fig. 8) and the level of enthusiasm seem to go hand in hand, i.e. the more difficult the material, the less enthusiasm experienced. In Racket, this trend is less visible. For example, animations (topic 3) are considered to be both challenging and inspiring at the same time. The lower levels of enthusiasm for the Python material could also be due to the fact that the material was originally meant for a different audience, whereas the Racket material was specifically designed for in-service math teachers. For instance, the math teachers did not find Python’s image processing particularly fit for their purposes. After the second module, the clamour for hands-on programming in order to learn became louder; yet during the first module, the Parson’s problems were regarded as both easy and motivating. Overall, the participants questioned the usefulness of some exercises because they lacked ready-made student material that could be utilized in a school context, which is probably also reflected in the enthusiasm scores. Below we summarize the most inspiring aspects of the courses in order. We also provide some excerpts from teachers’ responses (translated from Finnish).

Python

- Turtle images were fun and inspiring. *I was proud to be able to modify the Turtle code so that it formed a house*
- Image processing was inspiring as well. *As an art teacher it was easier to understand (than math)*
- Data Manipulation: Working with the Small Particles Data example (looking for values in the list, calculating averages), and the possibility to investigate real-life problems were beneficial

Suitable topics Fig. 10 represents the average suitability scores for each course topic (1 = not suitable at all, 5 = extremely suitable). The highest suitability score was given to Racket’s last topic, which includes material about CS as a new addition to the curriculum, CT, and pedagogical approaches to teaching the new content. In addition, the participants wrote an essay on ideas stimulated during the course and/or a lesson plan for integrating programming into their teaching. The next most suitable topics were Racket’s Image Programming (topic 1) and Turtle Graphics (topic 6). Topic 5 scored high as well, featuring a quiz that utilized recursive function and lists in implementation. The least suitable was topic 4, which employed recursion and user interaction.

Python scored notably lower in suitability. As mentioned, the material was not originally designed for math teachers, although these made up half of the participants (48 %). Hence, the generic material did not meet the teachers’ needs. Surprisingly in this regard, the Data Manipulation topic (number 5) averaged as the least suitable topic, even though it included real-life applications from, for instance, the statistics domain. Nevertheless, some teachers noted its value:

Python

- Turtle graphics could be used to teach geometry (focuses on angles and side lengths)
- Repetition, Decisions: using numbers, strings, turtles and images consistently within all topics was perceived positively
- Data manipulation (Statistics):
 - *In data analysis, real world problems made me understand what programming can be used for (environmental science and geography, analyzing air pollution in USA) and its usefulness in analyzing big amounts of data*
 - *Easy access to online data made me think I could use this in my teaching*
 - *Using statistics in mathematics could be useful (if I were a math teacher)*

Racket

- Image programming: drawing and designing own images. Possibility to see code, games and images drawn by other participants gave new ideas for teaching, i.e., sharing artifacts promoted creativity and increased enthusiasm

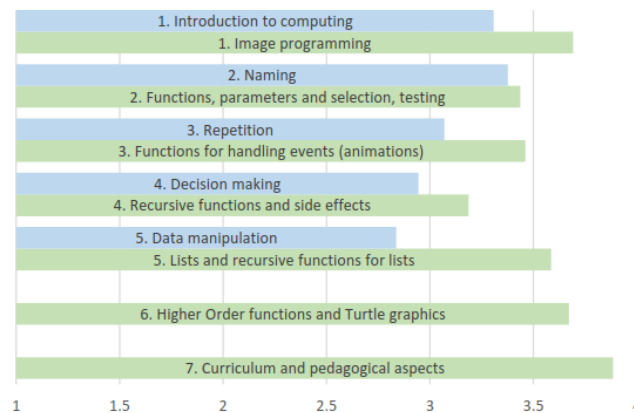


Fig. 10. Suitability of the course topics (P blue, R green)

Racket

- Possibility to utilize exercises in the school context
- Functions (Algebra): content was designed specifically for math teachers (including the exercises and utilizing the functional paradigm), thus the material and tools were useful for teaching

5.3 Which paradigms do the tracks align with and how do they support math?

Neither of the Code ABC tracks pronounced explicit paradigm considerations, nor were they present when the participants evaluated the topics. More experienced programmers compared languages – not paradigms – focusing mainly on the learning threshold (the lower the better) and differences in syntax, e.g. confusion with the excessive amount of parentheses and the prefix notation of Racket. However, several topics inherently implied paradigm-related issues. In the Python track, the Naming topic introduced data mutability and immutability, variable assignments, and in accordance with variables the most common misconceptions as well. The Python material highlighted the difference between functions and procedures, and introduced code division in the form of reusable modules and libraries. Python contains a class structure and an option for object-orientation but, in this context, Python was classified as imperative even though a few objects, such as turtles, were extensively utilized throughout the MOOC (in effect requiring the introduction of dot notation).

Similarly, the Racket material does not explicitly emphasize the underlying paradigm and hardly mentions the term ‘functional’ at all. However, the built-in principles of Felleisen’s ‘How to design a program’ [12] recommend avoiding imperative features and re-assignment of global variables. These principles aim at a purely functional programming style. However, the enhancement of `display-read` indicates the need for pointing out side-effects contradicting this purity: `display-read` interacts with a user. With regard to the variables in Racket, it is possible to define constants and `let` allows local variables to be assigned. These can still not be re-assigned without the `set!` command, forbidden by functional paradigm purists.

Having no re-assignable variables – thus no loop counters – has its implications for iterations as well. Although looping lists with the command of `foreach` is still possible, missing a re-assignable counter leads towards recursion and higher-order functions, where recursion calls “fake” mutability with expressions as function arguments and by returning partial results. Higher-order functions, such as `map`, `apply` or `filter`, creates new lists or accumulates values, based on given lists and functions. Later, these functional list-handling mechanisms were introduced in the Python language as well, along with lambda calculus and list comprehensions. In Python, however, there are “imperative” ways of looping (such as `for` and `while`), leaving a minimal need for recursion compared to Racket. In both tracks, the paradigm-influenced alternatives for implementing iteration and selection structures were the core content of the whole course.

As the paradigms reside implicitly in the material, it is no surprise that the teachers do not pay much attention to them. As with learning to drive a car, all attention is first drawn to the main aspects of driving – not to comparing the features of various car models. Nevertheless, teachers with previous programming experience compared the features of the course language to their previously learned languages. For instance, teachers in the Python track stated that *I learned that Python is simpler than Java (lists and their handling), I learned for and foreach loops and list modifications, and In my experience, Python requires less lines of code. Another nice thing was that when defining variables you don't need to think as much as in Java.*

In Racket, on the other hand, the teachers’ first impressions in particular were as follows: *I learned Racket's syntax. Writing mathematical expressions is cumbersome compared to Python, Java, Pascal, Ruby, and Visual Basic and If you consider using Racket at school, it is relatively complex compared to, for instance, Java. On a more positive note, Racket teachers also noted some benefits: Racket is really engaging! The first exercise was well selected: it is nice to immediately achieve some colorful shapes instead of the traditional “Hello World”. I did not know that programming can be this much fun!*

Algebra lies at the core in terms of math focus. It includes fundamentals such as functions, variables, statements and expressions, which are fundamental not only in math, but also in CS. In CS, functions and variables all encompass implementation mechanisms. However, the differences between concepts in these two disciplines may cause misconceptions and programming errors that are difficult to detect. In Table 1, module 2 compares the differences between functions in both tracks. Compared with Racket, Python allows remarkable freedom of implementation, which becomes particularly visible with functions.

In math, the function definition dictates at least one function parameter as input, and one and only one as output. It is possible to write a function without parameters in both languages, such as $k()$ (the rightmost figure in the 2nd row), which is not valid in math. In Python, a procedure $p(x)$ may return no value explicitly (in which case Python implicitly returns *None*), which is not acceptable in math, if x is in domain. Neither may a function return multiple values with the same input, as does the function $g(x)$, which returns either $y1$ or $y2$ based on the state.

What is wrong with the $f(x)$ on the Python side then? On the face of it, nothing: it gets at least one input as a parameter (in this case x) and returns only one output (y), as specified. However, the function body changes the global state of the outer program by re-assigning the global variable `stateChanged`, i.e., it causes a side-effect. Functions causing side-effects are not allowed in math. The idea of functional purity in the Racket track prevents side-effects, and in Python, no side-effects arise if the programmer is aware of the pitfalls and is capable of avoiding them. In accordance with a pure functional paradigm, immutable data and having no side-effects makes, for instance, parallel execution possible: different threads handling the same data can rely on the validity of the data.

In Racket, variables are essentially constants. In math, variables are also constants in the context of evaluating the value of an expression. The variables do not change during the evaluation, but between evaluations. For example, in the case of function $y = f(x)$, x changes when the position moves on the x-axis. Thus, in math, the term *variable* can be deceptive, because variable does not actually vary. Instead, the terms *a symbol* or *a representative* might give a more authentic view of the true meaning of the concept.

In the imperative paradigm, the situation with variables becomes even more obscure with the counter-type behavior. As an example, see the fourth row in Table 1, where the `while` loop exploits the value of i to decide when to stop. In the loop body, i is incremented with the assignment of $i = i + 1$. In math, this statement makes no sense. In CS, the statement is split by the equal sign to enable the left and the right value to be referred to separately. The left value opens a gaping rabbit hole to the underlying world of hardware specifics and constraints that are normally carefully guarded. In essence, it represents the memory address to which the right side – still a normal expression – is assigned.

Re-assignment is a dangerous operation and provides an endless source of error; for instance, if types are mixed in assignment. Thus, typing relates closely to variables. In static typing, a type must be given when a variable is defined and it is checked during the compilation. In dynamic typing, variables need not be assigned a specific type, but it may change assignment-by-assignment in runtime: now an integer, after the next assignment maybe a string. The operations allowed for integers differ remarkably from those allowed for strings. Strong typing prevents operations on invalid types [50], while prevention leads to compile and runtime errors when these are detected. Both Racket and Python are dynamically and strongly typed languages.

In Arithmetics, types of integer and decimal numbers provide more in-depth affordances. Typing relates to number sets in math, such as integers, and rational and irrational numbers. In math,

gradual progression from simple to more complex operations results in changes in the respective number sets as well. With addition and multiplication, a student remains in the comfort zone of integers. With subtraction, the student may move into negative numbers. A substantial paradigm shift happens when division is introduced, which along with fractions transfers a student into the zone of rational numbers, represented as decimals in code.

The ultimate challenge at elementary level is irrational numbers, which are met when a ratio is never-ending and non-repeating. Irrational numbers result e.g. from square root operations, and surds, such as $\sqrt{2}$, are never-ending. In a computer context, the limits of the physical memory allocated to each variable complicate the handling of such irrationalities. Historically in CS, selecting the right type has been important due to the constraints of memory size: a number has to be cut when the allocated bytes are used up resulting in the cut part being lost. The type implies the number of bytes in use, which influences the preciseness of a number.

Preciseness is also a consideration in many scientific calculations, for instance when rounding and defining significant numbers. In math, $\sqrt{2}$ or trigonometric expressions such as $\sin(60)$ are exact. However, when represented in decimal format they are not, irrespective of the length of the type of `float`. All in all, in Arithmetics, when calculating basic and advanced operations, a computer can be compared to a calculator, with which students practice new arithmetic functions such as `abs`, `sqrt`, or `exp`, and drill the right order of calculations.

In the Python material, math equations, such as speed-distance-time calculations, exemplify the use of a newly introduced mathematical functions. The Racket material is more geometry-oriented, placing greater emphasis on calculating areas, angles, and perimeters. Even if extensively used in examples, Geometry is not central to understanding the concepts of CS. Our study, however, notes its value as an area providing visually appealing applications.

Logic and logical operations (`and`, `or`, `not`) combine conditions into more complex conditions. Conditions – or logical expressions – fall into the area of logic. However, in the current math syllabus, logic does not have a prominent position. As logical expressions prompt control structures, the natural progression would be to learn logic first. Consequently, control structures and the use of conditions might fit within both Logic and Arithmetics.

Selections, or decisions as the Python track calls them, correspond to inequalities in math: 1D inequalities, such as $x > 0$, are represented on a number line. Whether the line is open-ended or close-ended depends on the comparison operator: `<` and `>` result in open-ended lines, while `≤` and `≥` result in close-ended ones. In 2D inequalities, such as $y > x$, the line divides the coordinate plane into two halves, one of which is shaded. An open-ended condition is represented as a dashed line, a closed condition as a solid one. Consequently, inequalities can be expanded to 3D as well.

Conditions can also be combined. When multiple inequalities hold simultaneously, the number-line is cut into segments, and lines define geometric shapes in a coordinate plane, most often triangles. In 3D, conditions may result in solid geometry shapes, such as prisms. In addition, conditions apply to piece-wise defined functions, which, for instance, have discontinuity points or behave differently depending on the range of x .

Iterations or recursive structures are relatively rare in elementary math. The accumulator pattern introduced in both tracks can be used in Statistics, e.g., when calculating mean values, or when looking for *min* or *max* values. Later, Σ and \prod operations are applied iteratively to the defined number domain and these new notations abbreviate e.g. the previous calculation of the mean. In Pre-Algebra, recognizing growing patterns prepares for sequences, abstracted later as functions in Algebra [57,56]. It is axiomatic that sequences and their sums and series are iterative. In inductive

problem solving, a student determines the n^{th} term in a sequence. Instead of the iterative `for` and `while` loops favoured by Python, Racket favours recursions. In contrast to induction, which starts from the first and aims at finding the n^{th} term, recursion starts from the n^{th} term and aims at reaching the first. The recursive calculation of factorial ($n!$) illustrates the product type iteration; see topic 4 in Table 1. Basic operations of Probability, combinations and permutations, make extensive use of factorials.

6 CONCLUSIONS

We have studied two approaches to teaching programming to in-service elementary school teachers in Finland. The majority of the participating teachers were mathematics teachers, which is understandable, given that programming has been added to the math syllabus in the new curriculum. Below we summarize our findings.

What CS concepts and CT skills do these Code ABC tracks introduce? Both tracks covered a substantial amount of basic programming concepts in their respective programming languages, such as subprograms (functions or procedures), conditional structures, boolean logic, data types, and lists. The Python track provided a generic synopsis of programming basics. As a primer, Python provided a history and some general knowledge about computers and CS. After this history review, the track focused on imperative programming fundamentals, such as assignment and `for/while` loops. Each new concept was demonstrated using numbers, strings, images, and Turtles. The Racket track, on the other hand, presented programming as yet another way of learning math rather than as a generic tool. Programming appeared as a new means of problem solving by exploiting functions. In addition to functions, control structures of selection and iteration were introduced. Iterations consisted of recursion and higher-order functions that manipulated lists.

CT links math and CS. When solving a problem, dividing the problem into smaller subproblems is essential (decomposition). In the context of programming, subproblems are subprograms, i.e. functions. Functions were discussed substantially more in Racket than in Python. As a recipe for a well-planned function, the Racket track introduced Design Recipe by Felleisen [12]. This Recipe promotes test-driven development: unit tests are implemented before a function body. Both courses emphasized the importance of using descriptive names for functions and variables, and the need for clear comments in order to improve readability; these coding conventions may be considered to be part of CT as well.

What topics did the teachers find challenging, inspiring, or suitable for math?

- Challenging:
 - The most challenging topics in the Python track were data and image processing, evidently due to the extensive exploitation of Repetition and Decision structures. Furthermore, the difficulty level suddenly rose when moving from Repetition to Decision.
 - In Racket, the most challenging topics included recursion (loops), animation, and lists, by far the most challenging of which was recursion.
 - The Racket track required a significant amount of effort because of the hands-on exercises and complex topics. Frequently, the exercises took more time than expected. This was experienced as a challenge by the participants, who had to take care of their normal work duties during the course.

- Inspiring (enthusiasm in the survey):
 - Turtle graphics were considered inspiring in both tracks. In Python, Turtle moves exemplified both Repetition and Decision topics, while in Racket, Turtle was linked with higher-order functions, which also rank high in the list of most challenging CS topics
 - In Python, the participants were interested in learning more about the history of CS, and the prominent and influential persons behind it. The Image Processing exercises divided opinion, some considered it interesting while others found it difficult, tedious, and a rather useless topic for the target group, (math teachers). Data processing was also received with mixed feelings. Some appreciated the real-life applications but a number of participants regarded it as too difficult.
 - The Racket participants valued highly creative and playful open-ended exercises allowing them to create their own 'art', even though this was not considered to be "traditional math" or central to conceptual learning. Sharing artifacts with others was one significant factor in creating enthusiasm and promoting creativity.
- Suitable for teaching:
 - In Python, suitability ranked significantly lower than in Racket. Even when the Racket track was challenging, it scored higher. The difference in suitability scores indicates that the course content should be tailored to better suit the target group, in this case math (and science) teachers.
 - In the Racket track, the participants regarded the pedagogical essay, image programming, Turtle graphics, animation and quiz as the most suitable and interesting.
 - Suitability and enthusiasm seem to correlate.

Which paradigms do the tracks align with and how do they support math? Conceptually, the functional paradigm is closer to math, in particular in its representation of functions and variables. The imperative paradigm comprises more elements that are foreign to pure math. As imperative, Python might call for less effort in its approach, but it contains built-in hazards that may cause misconceptions and programming errors. For instance, re-assigning a global variable changes the state and function outcome, thus conflicting with the mathematical definition of a function. In these paradigms, the meaning and importance of a variable varies as well. A variable's visibility is defined by its scope (local/global). In functional Racket, global variables are constant and re-assigning local variables is not advisable either. In Python, global variables can be re-assigned anywhere and types will change accordingly, which is indefensible from the viewpoint of math. Without having become used to re-assignable variables and the possibility of comparing, a novice programmer will learn the functional programming smoother and regard it more suitable and inspiring. In contrast, an experienced imperative programmer lacks his normal means of exploiting variables, which causes frustration.

The Finnish curriculum integrates CS into math without allocating more time to teach it. This necessitates making the CS syllabus as close as possible to math: no time can be wasted on learning irrelevancies or concepts causing unnecessary confusion. The curriculum, however, should determine the targeted CS concepts more precisely. Languages should be categorized based on those concepts and their math-suitability in order to make justified tool selections. Systematic research and various learning experiments will enable determination of the concepts, computational thinking skills, and teaching practices best suited to closing the digital skills gap, as stipulated by the Finnish Curriculum 2014.

7 ACKNOWLEDGMENTS

We gratefully acknowledge the grant support of and the Academy of Finland (grant number 303694; *Skills, education and the future of work*), the Finnish National Board of Education and Technology Industries of the Finland Centennial Foundation that enabled the research and the development of the Code ABC MOOC. In addition to the funders, we thank the Aalto University A+ and Rubyric teams for their efforts to continuously improve the MOOC platform. Last but not least, thanks to Tarmo Toikkanen, Tiina Korhonen, Otto Seppälä, and Arto Hellas for providing Code ABC MOOC material and corrections for this paper.

References

1. Alegre, F., Moreno, J.: Haskell in Middle and High School Mathematics. In: TFPiE. vol. 1. EPTCS, Sophia-Antipolis, France (2015)
2. Auvinen, T., Karavirta, V., Ahoniemi, T.: Rubyric: an online assessment tool for effortless authoring of personalized feedback. Workingpaper, ACM (2009)
3. Bal, H.E., Grune, D.: Programming language essentials. Addison-Wesley (1994)
4. Balanskat, A., Engelhart, K.: Computing our future: Computer programming and coding-Priorities, school curricula and initiatives across Europe (2014)
5. Barr, V., Guzdial, M.: Introducing CS to newcomers, and JES as a teaching tool. Communications of the ACM 59(11), 10–11 (2016)
6. Brady, Corey and Orton, Kai and Weintrop, David and Anton, Gabriella and Rodriguez, Sebastian and Wilensky, Uri: All Roads Lead to Computing: Making, Participatory Simulations, and Social Computing as Pathways to Computer Science. IEEE Transactions on Education 60(1), 59–66 (2017)
7. Burke, Q., Burke, Q.: Mind the metaphor: charting the rhetoric about introductory programming in K-12 schools. On the Horizon 24(3), 210–220 (2016)
8. Dijkstra, E.W.: How do we tell truths that might hurt? In: Selected Writings on Computing: A Personal Perspective, pp. 129–131. Springer (1982)
9. Ericson, B., Adrion, W.R., Fall, R., Guzdial, M.: State-Based Progress Towards Computer Science for All. ACM Inroads 7(4), 57–60 (2016)
10. Ericson, B., Guzdial, M., Morrison, B., Parker, M., Moldavan, M., Surasani, L.: An eBook for teachers learning CS principles. ACM Inroads 6(4), 84–86 (2015)
11. Ericson, B.J., Rogers, K., Parker, M., Morrison, B., Guzdial, M.: Identifying design principles for cs teacher ebooks through design-based research. In: Proceedings of the 2016 ACM Conference on International Computing Education Research. pp. 191–200. ICER '16, ACM, New York, NY, USA (2016), <http://doi.acm.org/10.1145/2960310.2960335>
12. Felleisen, M., Findler, R., Flatt, M., Krishnamurthi, S.: How to Design Programs, Second Edition. MIT-Press (2014), <http://www.ccs.neu.edu/home/matthias/HtDP2e/>
13. Felleisen, M., Krishnamurthi, S.: Viewpoint Why computer science doesn't matter. Communications of the ACM 52(7), 37–40 (2009)
14. Fesakis, G., Serafeim, K.: Influence of the familiarization with scratch on future teachers' opinions and attitudes about programming and ICT in education. In: ACM SIGCSE Bulletin. vol. 41, pp. 258–262. ACM (2009)
15. Finnish National Board of Education: National core curriculum for basic education 2014. In: National Core Curriculum for Basic Education 2014. Publications, Finnish National Board of Education (2016), <https://www.ellibs.com/fi/book/9789521362590/national-core-curriculum-for-basic-education-2014>
16. Futschek, G.: Algorithmic thinking: the key for understanding computer science. In: International Conference on Informatics in Secondary Schools-Evolution and Perspectives. pp. 159–168. Springer (2006)

17. Gagné, R.M.: *The Conditions of Learning*. New York: Holt, Rinehart and Winston (1965)
18. Gray, E.M., Tall, D.O.: Duality, ambiguity, and flexibility: A proceptual view of simple arithmetic. *Journal for research in Mathematics Education* pp. 116–140 (1994)
19. Guzdial, M.: Drumming up support for ap cs principles. *Communications of the ACM* 59(2), 12–13 (2016)
20. Guzdial, M., Soloway, E.: Computer science is more important than calculus: The challenge of living up to our potential. *SIGCSE Bulletin* 35(2), 5–8 (2003)
21. Guzdial, M.J., Ericson, B.: *Introduction to computing and programming in Python, a multimedia approach*. Prentice Hall Press (2009)
22. Gülbahar, Y., Kalelioglu, F.: The effects of teaching programming via Scratch on problem solving skills: A discussion from learners’ perspective. *Informatics in Education - An International Journal* 13.1(Vol13.1), 33–50 (2014)
23. Jarvis, S., Pavlenko, A.: *Crosslinguistic influence in language and cognition*. Routledge (2008)
24. Kiczales, G.: *UBCx: SPD1x Systematic Program Design - Part 1 (version 1, summer 2015)* (2015)
25. Kulik, J.A.: *Meta-analytic studies of findings on computer-based instruction, Technology assessment in education and training, vol. 1, pp. 9–34*. Psychology Press (1994)
26. Lee, I., Martin, F., Denner, J., Coulter, B., Allan, W., Erickson, J., Malyn-Smith, J., Werner, L.: Computational thinking for youth in practice. *ACM Inroads* 2(1), 32–37 (2011)
27. Lewis, C.M.: How programming environment shapes perception, learning and goals: Logo vs. Scratch. In: *Proceedings of the 41st ACM technical symposium on Computer science education*. pp. 346–350. ACM (2010)
28. Lutz, M.: *Learning Python: Powerful Object-Oriented Programming*. Safari Books Online, O’Reilly Media (2013), <https://books.google.fi/books?id=ePyeNz2Eoy8C>
29. Maloney, J., Resnick, M., Rusk, N., Silverman, B., Eastmond, E.: The Scratch programming language and environment. *ACM Transactions on Computing Education (TOCE)* 10(4), 16 (2010)
30. Marceau, G., Fisler, K., Krishnamurthi, S.: Measuring the effectiveness of error messages designed for novice programmers. In: *Proceedings of the 42nd ACM technical symposium on Computer science education*. pp. 499–504. ACM (2011)
31. Meerbaum-Salant, O., Armoni, M., Ben-Ari, M.: Learning computer science concepts with scratch. *Computer Science Education* 23(3), 239–264 (2013)
32. Monroy-Hernández, A., Resnick, M.: FEATURE empowering kids to create and share programmable media. *interactions* 15(2), 50–53 (2008)
33. Orni Meerbaum-Salant and Michal Armoni and Mordechai Ben-Ari: Habits of programming in scratch. In: *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education*. pp. 168–172. ACM (2011)
34. Papert, S.: An exploration in the space of mathematics educations. *International Journal of Computers for Mathematical Learning* 1(1), 95–123 (1996)
35. Papert, S., et al.: *Logo philosophy and implementation*. Logo Computer Systems Inc (1999)
36. Parsons, D., Haden, P.: Parson’s programming puzzles: a fun and effective learning tool for first programming courses. In: *Proceedings of the 8th Australasian Conference on Computing Education-Volume 52*. pp. 157–163. Australian Computer Society, Inc. (2006)
37. Partanen, T.: *Coding for schools: Coder’s Handbook (in Finnish)*. <http://racket.koodiaapinen.fi/manuaali/> (2016)
38. Partanen, T.: *Coding for schools: Student exercises (in Finnish)*. <http://racket.koodiaapinen.fi/tehtavat/> (2016)
39. Partanen, T., Mannila, L., Poranen, T.: Learning programming online: a racket-course for elementary school teachers in Finland. In: *Proceedings of the 16th Koli Calling International Conference on Computing Education Research*. pp. 178–179. ACM (2016)
40. Perkins, D.N., Salomon, G.: Teaching for transfer. *Educational leadership* 46(1), 22–32 (1988)
41. Reimann, P.: Design-based research, pp. 37–50. *Methodological choice and design*, Springer (2011)

42. Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B.: Scratch: programming for all. *Communications of the ACM* 52(11), 60–67 (2009)
43. Rich, P.J., Leatham, K.R., Wright, G.A.: Convergent cognition. *Instructional Science* 41(2), 431–453 (2013)
44. Robert W. Lent and Frederick G. Lopez and Kathleen J. Bieschke: Mathematics self-efficacy: Sources and relation to science-based career choice. *Journal of counseling psychology* 38(4), 424 (1991)
45. van Rossum, G.: Computer programming for everybody (1999)
46. Rossum, G.V.: Python Programming Language. In: *USENIX Annual Technical Conference*. vol. 41, p. 36 (2007)
47. Schanzer, E., Fisler, K., Krishnamurthi, S., Felleisen, M.: Transferring skills at solving word problems from computing to algebra through Bootstrap. In: *Proceedings of the 46th ACM Technical symposium on computer science education*. pp. 616–621. ACM (2015)
48. Schanzer, E., Fisler, K., Krishnamurthi, S., Felleisen, M.: Transferring skills at solving word problems from computing to algebra through Bootstrap. In: *Proceedings of the 46th ACM Technical symposium on computer science education*. pp. 616–621. ACM (2015)
49. Schanzer, E.T.: Algebraic Functions, Computer Programming, and the Challenge of Transfer (2015)
50. Scott, M.L.: *Programming language pragmatics*. Morgan Kaufmann (2000)
51. Sengupta, P., Kinnebrew, J.S., Basu, S., Biswas, G., Clark, D.: Integrating computational thinking with K-12 science education using agent-based computation: A theoretical framework. *Education and Information Technologies* 18(2), 351–380 (2013)
52. Toikkanen, T., Leinonen, T.: The Code ABC MOOC: Experiences from a Coding and Computational Thinking MOOC for Finnish Primary School Teachers. In: *Emerging Research, Practice, and Policy on Computational Thinking*, pp. 239–248. Springer (2017)
53. Van-Roy, P.: Programming paradigms for dummies: What every programmer should know. *New computational paradigms for computer music* 104 (2009)
54. Van-Roy, P., Haridi, S.: *Concepts, techniques, and models of computer programming*. MIT press (2004)
55. Wegner, P.: Guest editor’s introduction to special issue of computing surveys. *ACM Comput. Surv* 21, 253–258 (1989)
56. Wilkie, K.J.: Students’ use of variables and multiple representations in generalizing functional relationships prior to secondary school. *Educational Studies in Mathematics* pp. 1–29 (2016)
57. Wilkie, K.J., Clarke, D.M.: Developing students’ functional thinking in algebra through different visualisations of a growing pattern’s structure. *Mathematics Education Research Journal* 28(2), 223–243 (2016; 2015)
58. Wing, J.M.: Computational thinking. *Communications of the ACM* 49(3), 33–35 (2006)
59. Wing, J.M.: *Computational Thinking: What and Why?* Link Magazine (2010)
60. Wright, G., Rich, P., Lee, R.: The influence of teaching programming on learning mathematics. In: *Society for Information Technology & Teacher Education International Conference*. vol. 2013, pp. 4612–4615. editlib.org (2013)
61. Yoo, D., Schanzer, E., Krishnamurthi, S., Fisler, K.: WeScheme: the browser is your programming environment. In: *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education*. pp. 163–167. ACM (2011)
62. Zeldin, A.L., Pajares, F.: Against the odds: Self-efficacy beliefs of women in mathematical, scientific, and technological careers. *American Educational Research Journal* 37(1), 215–246 (2000)