Constructionist Attempts at Supporting the Learning of Computer Programming: A Survey

Michael Lodi, Dario Malchiodi, Mattia Monga, Anna Morpurgo, Bernadette Spieler

Constructionist attempts at supporting the learning of computer programming: a survey[1]

Michael Lodi • Alma Mater Studiorum - Università di Bologna & INRIA Focus, Italy • michael.lodi/at/unibo.it
Dario Malchiodi, Mattia Monga, Anna Morpurgo • Università degli Studi di Milano, Italy • {malchiodi, monga, morpurgo}/at/di.unimi.it
Bernadette Spieler • Technische Universität Graz, Austria • bernadette.spieler/a/ist.tugraz.at

Abstract

Although programming is often seen as a key element of constructionist approaches, the research on learning to program through a constructionist strategy is somewhat limited, mostly focusing on how to bring the abstract and formal nature of programming languages into "concrete", possibly tangible objects, graspable even by children with limited abstraction power. We survey the literature in programming education and analyse some programming languages designed to help novices from a constructionist perspective.

Introduction

While programming is often seen as a key element of constructionist[2] approaches (starting from LOGO (Feuerzeig et al. 1970), a programming language designed to enable learning abstract concepts of disciplines like math, geometry, physics, and potentially all others, by manipulating computational objects (Papert 1980)), the research on learning to program through a constructionist strategy is somewhat limited, mostly focusing on how to bring the abstract and formal nature of programming languages into "concrete" or even tangible objects, accessible also to children with limited abstraction power (Resnick et al. 2009; Kay et al. 1997; Horn and Jacob 2007; Dann et al. 2008; Hauswirth, Adamoli, and Azadmanesh 2017). Notwithstanding this, programming is in some sense intrinsically constructionist, as it always involves the production of an artifact that can be shown and shared. Of course, this does not mean that programming automatically leads to constructivist/constructionist pedagogies: in facts, we see very different approaches, from open project-based learning to much more traditional education through lectures and closed exercises. Specific languages and

[1] This is an authors' pre-print version of the work. It is posted here for your personal use. Not for redistribution. The definitive version was published in Olympiads in Informatics 13 (2019), 99–121. https://ioinformatics.org/journal/v13_2019_99_122.pdf

[2] Constructionism originated from Seymour Papert, drawing on Jean Piaget's constructivist view that knowledge needs to be (re)constructed rather than transmitted (Piaget, 1973), and adding that this is particularly effective when involves the construction of a (concrete or abstract) artifact, meaningful for the learner (Papert 1980).

environments play an important role too: for example, visual programming languages make it easier (by removing the request to face unnatural textual syntactic rules) to realize small but meaningful projects, keeping students motivated, and support a constructionist approach where students are encouraged to develop and share their projects — video games, animated stories, or simulations of simple real-world phenomena. Constructionist ideas are also floating around mainstream programming practice and they are even codified in some software engineering approaches: agile methods like eXtreme Programming (Beck and Andres 2004), for example, suggest several techniques that can be easily connected to the constructionist word of advice about discussing, sharing, and productively collaborating to successfully build knowledge together (Resnick 1996); moreover the incremental and iterative process of creative thinking and learning (Resnick 2007) fits well with the agile preference to "responding to change over following a plan" (Beck et al. 2001). It actually originated by observing how the traditional kindergarten approach to learning is ideally suited to learn to think creatively, and it is now called "creative learning spiral" (Figure 1). According to this model, when one learns by creating something (*e.g.,* a computer program) she *imagines* what she wants to do, *creates* a project based on this idea, *plays* with her creation, *shares* her idea and her creation with others, *reflects* on the experience and feedback received from others, and all this leads her to *imagine* new ideas, new functionalities, new improvements for her project, or new projects. The process is iterated many times. This spiral describes an iterative process, highly overlapping with the iterative software development cycle.

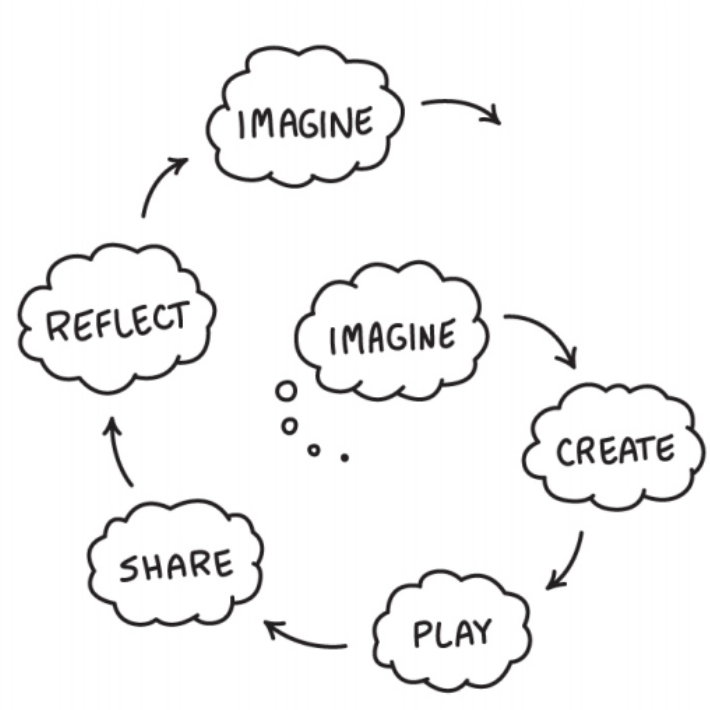

Figure 1: Creative learning spiral (source: (Resnick 2017)).

What does it mean to learn programming?

The basic premise behind programming — *i.e.,* producing a precise description of how to carry out a task or to solve a problem — is that an *interpreter*, different from the producer of the description, can understand it and effectively carry out the task as described. There are thus two distinct but tightly tied aspects in programming:

i. the program itself (the text or other streams of symbols or actions that build up the digital coding of an algorithm),
ii. the actions that take place when the program is run by the interpreter.

This distinction is explicit in most of the professional programming environments, but it is conceptually present even in those environments designed for very small children, where the program is somewhat implicit. The Bee-Bot[3], for example, is a bee-shaped robot that can be programmed by pushing the buttons on its back: the program, while recorded and then executed by the machine, is not explicit nor visible in its static form by the children, but it exists, and the programmer needs to master the relationship between the actions she records into the bee and the actions the bee will perform when the program will be executed. In this paper, however, we focus on programs in which the source code is explicit, as it is common in programming activities proposed to secondary school pupils.

Thus, one needs to know the interpreter in order to program, in particular:

- the set of basic actions it is able to perform,
- the language it is able to understand, with rules on how to compose basic actions,
- the relation between *syntax* and *semantics,* that is what actions it will perform given a description, and, conversely, how to describe a given sequence of actions so that it will perform them.

The first aspect, that is the program *source code,* is explicit, visible. The second one instead, that is the actions that take place when the program is run, is somewhat implicit, hidden in the execution time world, and not so immediate to grasp for novices. Moreover, this aspect is sometimes underestimated by both teachers and learners: teachers, as experts, give it for granted; learners tend to construct personal intuitive, not necessarily coherent, ideas of what will happen.
This dichotomy of programming — its static visible code and its implicit dynamics — emerges as a critical issue when learning to program, as shown by studies from different perspectives. To cite a few (Sorva 2013):

- Phenomenography studies show how novice programmers tend to perceive programming as no more than the production of code, missing to relating instructions in the program to what happens when the program is executed.
- Studies on programming misconceptions point out how most of programming misconceptions have to do with aspects that are not readily visible in the code but are related to the execution time, both in term of what will happen and of what will not unless explicitly specified in the code.
- Threshold concept theory identifies program dynamics as a candidate threshold concept in programming as it has many of the features that characterize threshold concepts; among others: it is a troublesome barrier to student understanding, it

[3] https://www.bee-bot.us/

transforms how the student perceives the subject, it marks a boundary between programmers and end users.

To help novice programmers take into account also the dynamic side of programming, the concept of *notional machine* (Du Boulay 1986; Sorva 2013) has been proposed. A notional machine is a characterisation of the computer in its role as executor of programs in a particular language (or set of languages, or even a subset of a language) for didactic purposes. It thus gives a convenient description of the association syntax-semantics.

The following learning outcomes should therefore be considered when teaching to program:

- the development by students of a perception of programming that does not reduce to the production of code, but includes relating instructions to what will happen when the program is executed, and eventually comes to include producing applications for use and seeing it as a way to solve problems;
- the development of a mental model of a notional machine that allows them to make the association (static) syntax - (dynamic) semantics and to trace program execution correctly and coherently.

In particular, this latter outcome goal will include the development of the following skills:

- given a program (typically one's own) and an observed behaviour:
    - identify when debugging is needed because the behaviour is somewhat not the one intended,
    - identify where a bug has occurred,
    - be able to correct the code;
- given a program and its specification, be able to test it;
- understand that there can be multiple correct ways to program a solution.

If these are crucial points in learning to write executable descriptions, however, programming is indeed a multifaceted competence, and the knowledge to construct and the skills to develop span over several dimensions, besides predicting concrete semantics of abstract descriptions. A skilled programmer needs to:

1. understand general properties of automatic interpreters able to manipulate digital information;
2. think about problems in a way suitable to automatic elaboration;
3. devise, analyze, compare solutions;
4. adapt solutions to emerging hurdles and needs;
5. integrate into teamwork and be able to elicit, organize, and share the abstract knowledge related to a software project.

Here we mainly focus on skill 1 and the support provided by programming languages and environments. Moreover we highlight the opportunity provided by agile methodologies to develop skill 5.

Unplugged activities

Offline or *unplugged* programming activities have often been used to explain important concepts or vocabulary to students without actually using a PC, laptop, or smartphone,

e.g., x/y coordinates, the need for precise instructions for computers/robots, or variables and lists. Examples are to program a classmate like a robot, give paint instructions, pack a rucksack, or send "broadcast messages" to colleagues.
Unplugged activities in small groups have become popular over the years to introduce basic computer science concepts in non-vocational contexts. They offer:

a constructivist environment: indeed
- by manipulating real objects or dramatising processes, pupils can observe what happens, formulate hypotheses, validate them through experiments, i.e. develop a scientific approach to the construction of their knowledge;
- by working in a group, pupils are encouraged to participate, share ideas, verbalize and uphold their deductions.

inexpensive set up: they usually require very basic and inexpensive materials, so they can be easily proposed in different contexts;

no technological hurdles: they allow students (and teachers) to have meaningful experiences related to important CS concepts (like algorithms) without having to wait until they get some technology and programming fluency (Bell and Lodi, to appear).

It is important to note that evidence shows unplugged activities should not replace programming activities, but can be helpful to make them more effective (Bell and Vahrenhold 2018).
The following two examples, taken from CS Unplugged[4] and ALaDDIn[5], illustrate typical unplugged approaches to introduce children to programming.
In CS Unplugged "Rescue Mission", pupils are given by the teacher a very simple language with only three commands: 1 step forward, 90 degrees left, 90 degrees right. The task is to compose a sequence of instructions to move a robot from one given cell on a grid to a given other cell. Pupils are divided into groups of three where each one has a role: either programmer, bot, or tester. This division of roles is done to emphasize the fact that programs cannot be adjusted on the fly; they must be first planned, then implemented, then tested and debugged until they work correctly.
ALaDDIn "Algomotricity and Mazes" is an activity designed according to a strategy called algomotricity (Lonati et al. 2011; Bellettini et al. 2012, 2013, 2014), where pupils are exposed to an informatic concept/process by playful activities which involve a mix of tangible and abstract object manipulations; they can investigate it firsthand, make hypotheses that can then be tested in a guided context during the activity, and eventually construct viable mental models. Algomotricity starts "unplugged" (Bell, Rosamond, and Casey 2012) but ends with a computer-based phase to close the loop with pupils' previous acquaintance with applications (Taub, Armoni, and Ben-Ari 2012).
"Algomotricity and Mazes" focuses on primitives and control structures. The task is that of verbally guiding a "robot" (a blindfolded person) through a simple path. Working in groups, pupils are requested to propose a very limited set of primitives to be written each on a sticky note, and to compose them into a program to be executed by the "robot". Also, they have the possibility of exploiting basic control structures (if, repeat-until, repeat-n-times). The conductor may decide to swap some programs and "robots", in order to emphasize the ambiguity of some instructions or the dependency of programs on special features of the "robot" (*e.g.,* step/foot size). In the last phase, students are given computers and a slightly modified version of Scratch. They are requested to write programs that guide a sprite through mazes of increasing complexity

[4] https://csunplugged.org/
[5] http://aladdin.di.unimi.it/

where shape patterns foster the use of loops.

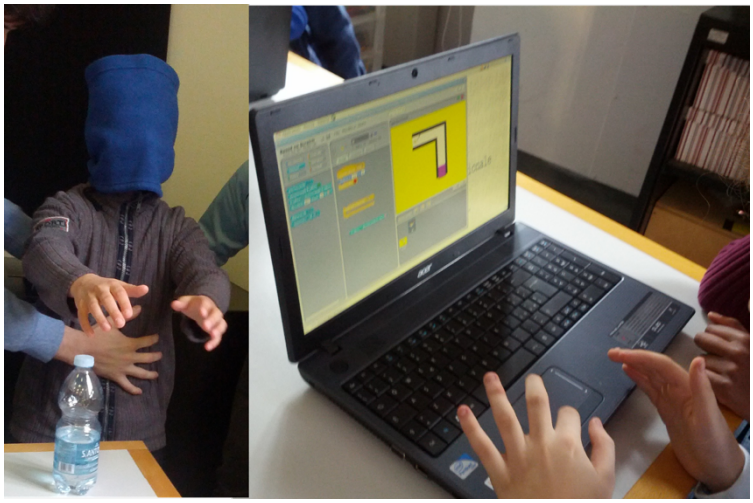

Figure 2: The first and last phase of the "Algomotricity and Mazes" activity, respectively.

Notional machines

An important intuition for approaching programming from a constructionist perspective is that programs are a join point between our mind and the computer, the interpreter of the formal description of what we have in mind. Thus, programs appeal to our curiosity and ingenuity and are wonderful artifacts to share and discuss with other active minds. Such a sharing, however, assumes that the interpreter is a shared knowledge among peers. When a group of people programs the same 'machine', a shared *semantics* is in fact given, but unfortunately people, especially novices, do not necessarily write their programs for the formal interpreter they use, rather for the *notional machine* (Sorva 2013; Berry and Kölling 2014) they actually have in their minds.

A notional machine is an abstract computer responsible for executing programs of a particular kind (Sorva 2013) and its grasping refers to all the general properties of the machine that one is learning to control (Du Boulay 1986). The purpose of a notional machine is to explain, to give intuitive meaning to the code a programmer writes. It normally encompasses an idealized version of the interpreter and other aspects of the development and run-time environment; moreover, it should bring also a complementary intuition of what the notional machine cannot do, at least without specific directions of the programmer.

To introduce a notional machine to the students is often the initial role of the instructors. Ideally this should be somewhat incremental in complexity, but not all programming languages are suitable for incremental models: in fact, most of the success for introductory courses of visual languages or Lisp dialects is that they allow shallow presentations of syntax, thus letting the learners focus on the more relevant parts of their notional machines.

An explicit reference to the notional machine can foster meta-cognition and, during teamwork, it can help in identifying misconceptions. But how can the notional machine be made explicit? Tracing of the computational process and visualization of the execution are effective candidate tools. They allow instructors to make as clear as possible: i) what novice programmers should expect the notional machine will do and ii) what it actually does.

Abstract programming patterns

A small number of abstract programming patterns can be applied to a potentially infinite spectrum of specific conditions. This is often a challenge for novices, given that most of the times the discipline is taught (i) introducing one or more primitive tools (e.g., variables), and (ii) showing some examples highlighting how these tools can be used to solve specific problems. This might lead to the rise of *misconceptions* of pupils w.r.t. the above-mentioned tools.

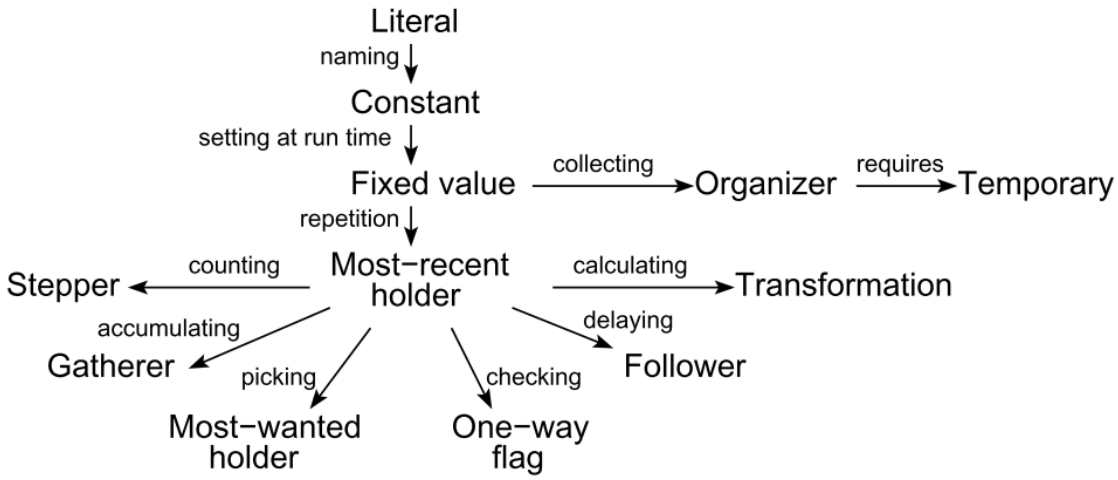

Figure 3: Roles of variables, organized in a constructionist-like hierarchy where the predecessor of an arrow is a prerequisite for learning the corresponding successor (source: (Sajaniemi 2002)).

The concept of *role of variables* (Sajaniemi 2002; Proulx 2000) has been proposed in order to guide novice programmers from the operational knowledge of a variable as the holder of a mutable value to the ability to identify *abstract* use cases following a small number of roles (such as those in Figure 3). Such ability is of great help when tackling the solution of a specific problem, for instance, that of computing the maximal value within a sequence. Indeed, this is a great opportunity for letting pupils realize that this problem is a special case of the more general quest for optimal value. The latter can be found using a *most-wanted holder* to be compared with each element of the sequence and containing the highest value seen so far. This method easily fits the search of the maximal as well as the minimal value, and it also efficiently handles less obvious cases such as that of finding the distinct vowels occurring in a sentence.

These roles can also be gradually introduced following the hierarchy of Figure 3, starting from the concept of literal (e.g., an integer value or a string) and building knowledge about one role on the top of already understood roles.

For selection and iteration as well there are several standard use patterns that occur over and over again. Selection patterns (Bergin 1999) and loop patterns (Astrachan and Wallingford 1998) have been introduced with the same goal. For instance, to illustrate the idea, the *loop and a half* pattern is an efficient processing strategy for a sequence of elements whose end can be detected only *after* at least one element has been read. It uses an infinite loop whose body accesses the next sequence element. If there are no more elements, the loop is escaped through a controlled jump, otherwise some special actions are possibly executed before continuing the iteration. Figure 4 shows one of the canonical incarnations of this pattern: the possibly repeated check of a value given as

input, detecting and ignoring invalid entries.
Selection and loop patterns fit well within a constructionist-based learning path: they might be *naturally* discovered when critically analyzing software implementations. For instance, the previous loop could be the end point of a reasoning scheme started from the detection of a duplicated line of code in a quick-and-dirty initial implementation.

```
while True:
    value = input('insert a positive, odd value')
    if value > 0 and value % 2 == 1:
        break
    print('the value is not valid')
```

Figure 4: A typical loop and a half pattern applied to the repeated validation of external inputs to a procedure

In general, abstract programming patterns are provided in a short number, in order to cover them within a standard introductory computer programming course; moreover, the related concepts are easily grasped by experienced computer science teachers (Ben-Ari and Sajaniemi 2004), thus they can be embedded in already existing curricula with low effort.

Misconceptions

Sorva defines *misconceptions* as "*understandings that are deficient or inadequate for many practical programming contexts"* (Sorva 2013).
Some authors (Ben-Ari 2001) believe that computer science has an exceptional position in constructivist's view of knowledge constructed by individuals or groups rather than a copy of an ontological reality: in fact, the computer forms an "accessible ontological reality" and programming *features many concepts that are precisely defined and implemented within technical systems [...] sometimes a novice programmer* "*doesn't get*" *a concept or* "*gets it wrong*" *in a way that is not a harmless (or desirable) alternative interpretation. Incorrect and incomplete understandings of programming concepts result in unproductive programming behavior and dysfunctional programs* (Sorva 2013).
According to Clancy (2004), there are two macro-causes of misconceptions: *over- or under-generalizing* and *a confused computational model*. High-level languages provide an abstraction on control and data, making programming simpler and more powerful, but, by contrast, hiding details of the executor to the user, who can consequently find mysterious some constructs and behaviors.
Much literature about misconceptions in CSEd can be found: we list some of the most important causes of misconceptions, experienced especially by novices, divided into different areas, found mainly in (Clancy 2004; Sirkiä 2012; Sorva 2013) and in the works they reference. For a complete review see for example (Qian and Lehman 2017).

English Keywords of a language do not have the same meaning in English and programming. For example, the word *while* in English indicates a constantly active test, while the construct `while` can test the condition again only at the beginning of the next iteration. Some students believe that the loop ends at the precise moment the condition is falsified. Similarly, some of them think of the `if` construct as a test continuously active and awaiting the occurrence of a condition, others believed that the `then` branch is executed as soon as the condition becomes true.

Syntax Although one may think the syntax is one of the biggest sources of misconceptions, studies show that it is a problem only in the very early stages. In particular, some students were able to write syntactically valid programs, which, however, were not useful for solving the given problem, or were semantically incorrect.

Mathematical notation Reported by many authors, classical is the confusion that generates the assignment with the = symbol (for example, seen as an equation or as a swap of values between variables) or the increment (`a = a + 1`) thought of as an impossible equation.

Examples of over-generalization Some authors found a series of non-existent constraints (e.g., methods in different classes that must have different names, arguments that can only be numbers, "dot" operator usable just in methods) dictated by the fact that the students had not seen any counterexample for such situations.

Similarities The analogy "a variable is like a box" can foster the idea that - like a box - it can contain more elements at the same time. The analogy "programming with the computer is like conversing with it" can bring to attribute *intentionality* to the computer and therefore to think that it:

- has a hidden intelligence that understands the intentions of the programmer and helps her achieve her goal (the so-called "superbug");
- has a general vision, knowing also what will happen in lines of code that it is not currently running.

Some aspects of programming are particular carriers of misconceptions.

Sequence Many misconceptions are due to lack of understanding of the program flow: all lines active at the same time, "magic" parallelism, the unimportance of the order of instructions, difficulty in understanding the branches.

Passing parameters Students present difficulties in this area, for example by confusing the types of passing (by value, by reference, ...), making mistakes with the return value or with the parameters' scope.

Input Input statements are particularly problematic. Students do not understand where the input data come from, how they are stored and made available to the program. Some of them believe that a program remembers all the values associated with a variable (its "history").

Memory allocation There are considerable difficulties in understanding the memory model of languages where allocation happens implicitly.

Programming languages for learning to program

From a constructionist viewpoint of learning, programming languages have a major role: they are a key means for sharing artifacts and expressing one's theories of the world. The crucial part is that artifacts can *be executed* independently from the creator: someone's (coded) mental process can become part of the experience of others, and thus criticized, improved, or adapted to a new project. In fact, the origin of the notion itself of constructionism goes back to Papert's experiments with a programming environment (LOGO) designed exactly to let pupils tinker with math and geometry (Papert 1980). Does this strategy work even when the learning objective is the programming activity itself? Can a generic programming language be used to give a concrete reification of the computational thinking of a novice programmer? Or do we need something specifically designed for this activity? Alan Kay says that programming languages can be

categorized in two classes: “agglutination of features” or “crystallization of style” (Kay 1993). What is more important for learning effectively in a constructivist way? Features or style?

In the last decade, a number of block-based programming tools have been introduced to help students have an easier time when first practicing programming. These tools, often based on web-based technologies, as well as an increase in the number of smartphones and tablets, opened up new ways for innovative coding concepts (Kahn 2017). In general, they focus on younger learners, support novices in their first programming steps, can be used in informal learning situations, and provide a visual language which allows students to recognize blocks instead of recalling syntax (Tumlin 2017). Many popular efforts for spreading computer science in schools, like (Goode, Chapman, and Margolis 2012) or the teaching material from Code.org,[6] rely on the use of such environments. In addition, such tools have been adopted into many computing classes all over the world (Meerbaum-Salant, Armoni, and Ben-Ari 2010).

LOGO

LOGO was designed (since 1967) for (constructionist) educational purposes by Wally Feurzeig, Seymour Papert, Cynthia Solomon, Daniel Bobrow, Richard Grant (Papert 1980). Its syntax was heavily influenced by Lisp (at the time the standard language for Artificial Intelligence research) and it was initially designed to aid students in learning secondary school mathematics. The most successful LOGO version featured a graphical (at least in principle) environment: instructions are directed to a “turtle” who moves around the screen, possibly leaving a colored trace. The turtle should help learners (especially the younger ones) with a sort of self-identification: its movements have a clear correspondence with their movements in the real world. The patterns drawn by the turtle can be the way the learners build their understanding of 2D geometry, discovering in the process even deep mathematical truths as the fact that a circle can be approximated by a high number of straight segments (Abelson and DiSessa 1986) (see Figure 5).

```
TO CIRCLE
   REPEAT FOREVER
      [
         FORWARD 1
         RIGHT 1
      ]
```

Figure 5: A procedure to draw a circle in LOGO

Interestingly enough, LOGO was originally conceived to empower learners of mathematics/geometry, not programming. Programming is *just* a means of expression, but one with great epistemic potential. According to Papert: “in teaching the computer how to think, children embark on an exploration about how they themselves think. The experience can be heady: Thinking about thinking turns every child into an epistemologist, an experience not even shared by most adults” (Papert 1980). Also, by expressing something in a way the LOGO turtle can “understand” can be fruitful for real-world activities, too. Juggling, for example, can be analyzed with LOGO: the identification of *proper* sub-activities (*i.e.,* sub-routines like TOP-RIGHT to recognize when one juggling ball is at the top of its trajectory going to the right, or TOSS-LEFT to

[6] https://hourofcode.com

throw the ball with the left hand) may shorten significantly the time for acquiring juggling skills (from days to hours, according to (Papert 1980)). And here 'proper' should be understood as appropriate to the task, but also as "fitting properly with the programming language idiomatic way of describing computational processes". LOGO had many independent implementations and its approach is still very popular, even Python has a `turtle` package in its standard library.

Smalltalk

Smalltalk (Goldberg and Kay 1976) also has its roots in constructionist learning. Back in the early seventies, at the Learning Research Group within the Xerox Parc Research Center, people were envisioning a world of personal computing devices which should have "programmability". Smalltalk, whose lineage traces clearly to LOGO and Lisp, was designed with a general audience in mind, since everyone should be comfortable with programming and computing devices should become ubiquitous in learning environments "along the lines of Montessori and Bruner" (Kay 1993). Thus, Smalltalk was not directed specifically to children and it has conquered a wide professional audience. In Smalltalk everything is an 'object' able to react to 'messages'. It follows a highly consistent object-oriented approach and code can be factored out by inheritance and dynamic binding. Smalltalk introduces also the idea that everything in the system is programmable: such a dynamic environment encourages a trial-and-error approach. A specific Smalltalk system for children was designed later as an evolution of Squeak Smalltalk: E-toys (Kay et al. 1997) provided a world of "sprites", funny characters that can be moved (concurrently) around the screen by programming them in Smalltalk. E-toys then evolved in Scratch, where the programming part was replaced by visual blocks.

BASIC, Pascal

It seems legitimate to mention BASIC (Beginner's All-purpose Symbolic Instruction Code (Kurtz 1978)) in a paper on constructionism and programming: for years BASIC has been the elective language for personal projects and even before widespread Internet connectivity, several communities shared BASIC programs in Bulletin Board Systems and magazines. Its popularity among self-taught programmers, however, was due mainly to its availability on personal and home computing devices. Moreover, the language was typically implemented using an interpreter, thus naturally fostering the trial-and-error and incremental learning styles typical of a constructionist setting. A generation grown with BASIC still thinks it is a wonderful approach to get children hooked on programming (see for example (Brin 2016)). However, many believe BASIC is not able to foster good abstractions and fear that BASIC programmers will bring bad habits to all their future computational activities.
In 1970 Niklaus Wirth published Pascal (Wirth 1993), a small, efficient language intended to encourage sound programming practices using structured programming and data structuring. For about 25 years, Pascal (and its successors like TurboPascal or Modula-2) was the most popular choice for undergraduate courses and a whole generation of computer scientist learned to program through its discipline popularized by Wirth in his book "Algorithms + Data Structures = Programs". Only Java had similar success in undergraduate courses. However, while Java popularity was (and is) influenced by trends in the software industry, Pascal was appealing mainly for its intrinsic discipline, which matched the academic sentiment of the time.

Scheme, Racket

Scheme (Abelson et al. 1998) is a language originally aimed at bringing structured

programming in the lands of Lisp (mainly by adding lexical scoping). The language has nowadays a wide and energetic community of users. Its importance in education, however, is chiefly related to a book, "Structure and Interpretation of Computer Programs" (SICP) (Abelson, Sussman, and Sussman 1996), which had a tremendous impact on the practice of programming education. The book derived from a semester course taught at MIT. It has the peculiarity to present programming as a way of organizing thinking and problem solving. Every detail of the Scheme notional machine is worked out in the book: at the end, the reader should be able to understand the mechanics of a Scheme interpreter and to program one by herself (in Scheme). The book, which enjoyed widespread adoption, was originally directed to MIT undergraduates and it is certainly not suitable either for children or even adults without a scientific background: examples are often taken from college-level mathematics and physics.

A spin-off of SICP explicitly directed to learning is Racket. Born as 'PLT Scheme', one of its strength is the programming environment DrScheme (Findler et al. 2002) (now DrRacket): it supports educational scaffolding, it suggests proper documentation, and it can use different *flavours* of the language, starting from a very basic one (Beginning Student Language, it includes only notation for function definitions, function applications, and conditional expressions) to multi-paradigm dialects. The DrRacket approach is supported by an online book "How to design programs" (HTDP) [7] and it has been adapted to other mainstream languages, like Java (Allen, Cartwright, and Stoler 2002) and Python. The availability of different languages directed to the progression of learning should help in overcoming what the DrRacket proponents identify as "the crucial problem" in the interaction between the learner and the programming environment: beginners make mistakes before they know much of the language, but development tools yet diagnose these errors as if the programmer already knew the whole notional machine. Moreover, DrRacket has a minimal interface aimed at not confusing novices, with just two simple interactive panes: a definitions area, and an interactions area, which allows a programmer to ask for the evaluation of expressions that may refer to the definitions. Similarly to what happens in visual languages, Racket allows for direct manipulation of sprites, see an example in Figure 6.

```
(define (picture-of-rocket.v3 height)
  (cond
    [(<= height (- 60 (/ (image-height  ) 2)))
     (place-image  50 height
                   (empty-scene 100 60))]
    [(> height (- 60 (/ (image-height  ) 2)))
     (place-image  50 (- 60 (/ (image-height  ) 2))
                   (empty-scene 100 60))]))
```

Figure 6: Racket code for "landing a rocket"

[7] Current version: http://www.htdp.org/2018-01-06/Book/index.html

The authors of HTDP claim that "program design — but not programming — deserves the same role in a liberal arts education as mathematics and language skills." They aim at systematically designed programs thanks to systematic thought, planning, and understanding from the very beginning, at every stage, and for every step. To this end, the HTDP approach is to present "design recipes", supported by predefined scaffolding that should be iteratively refined to match the problem at hand. This is indeed very close to the idea of micropatterns discussed above.

Scratch, Snap!, Alice, and others

EToys worlds with pre-defined — although programmable — objects, evolved in a generic environment in which everything can be defined in terms of 'statement' blocks. Scratch (Resnick et al. 2009), originally written in Smalltalk, is the most popular and successful visual block-based programming environment. Launched in 2007 by the MIT Media Lab, the Scratch site has grown to more than 25 million registered members with over 29 million Scratch projects shared programs.
Unlike traditional programming languages, here graphical programming blocks are used that automatically snap together like Lego bricks when they make syntactical sense (Ford 2009). In visual programming languages, a block represents a command or action and they are arranged in scripts. The composition of individual scripts equals the construction of an algorithm. The building blocks offer the possibility, *e.g.,* to animate different objects on a stage, thus defining their behavior.
The Scratch environment has some distinctive characteristics, according to its authors (Maloney et al. 2010). Among the ones the authors highlight, some are particularly relevant in the constructionist approach:

Liveness The code is constantly running and can be changed on the fly, immediately seeing the runtime effects of the change; this encourages users to tinker with the code.

No error messages When you play with Lego bricks, they stack together or they don't - the same happens in Scratch; program always run: syntax errors are prevented from the block shapes and connections, and also runtime errors are avoided by doing something "reasonable" (*e.g.*, in the case of an out-of-range value); this is particularly important not to frustrate kids and to keep them iterating and developing: "*A program that runs, even if it is not correct, feels closer to working than a program that does not run (or compile) at all*" (Maloney et al. 2010).

Other characteristics are useful to help novices avoiding misconceptions that often arise when starting to learn to program.

Execution made visible A glowing yellow border surrounds running scripts (in some versions each block is highlighted when it is executed); this is very helpful in program reading and debugging, and helps students form a correct mental model of the notional machine underlying the program execution.

Making data concrete You can see in a variable box, automatically shown, its current value: again, this is helpful for making the underlying machine model visible.

Finally, other characteristics introduce important software engineering and development concepts.

Open source Each shared project has a "see inside" button that brings you to the project source; you can read and edit the blocks to see what happens.

Remixing If you edit someone else's project, you create a remix: you are the author,

but the system automatically gives credits to the original author (at any depth, keeping track of multiple remixes in a tree) and suggests you to explicitly declare what changes you made.

The main limitation of Scratch programs is that they do not scale well from the abstraction point of view: only since version 2 you can "make a new block" that is, a procedure with optional parameters. These blocks have no possibility to return a value (like a number or a boolean) and so can't be nested inside other blocks, forcing you to modify global variables if needed.

Snap![8] (originally BYOB, Build Your Own Blocks) is an extended reimplementation of Scratch with functions and continuations. These added capabilities make it suitable for a serious introduction to computer science for high school or college students: in fact, Snap! is used as the basis for an Advanced Placement CS course at Berkeley[9].

The Scratch approach was also ported to mainstream programming languages: in Alice (Dann, Cooper, and Pausch 2008) visual blocks are in fact Java instructions. Alice worlds are 3D: this choice makes it very attractive and appealing to pupils (Rodger et al., 2009), who can program amazing 3D animations. It also adds many complexities, since moving objects in a 3D space is not trivial.

Recently, these environments evolved towards web or phone/tablet versions, in order to be available in the contexts more popular within young people. For example, Pocket Code[10] allows the creation of games, stories, animations, and many types of other apps directly on phones or tablets, thereby teaching fundamental programming skills (Slany, 2014). In some cases block and textual programming languages are interchangeable. In many cases these environments can connect to physical devices and sensors, with the goal of increasing the constructionist appeal of block programming, and opening to the world of "tinkering" with electronics.

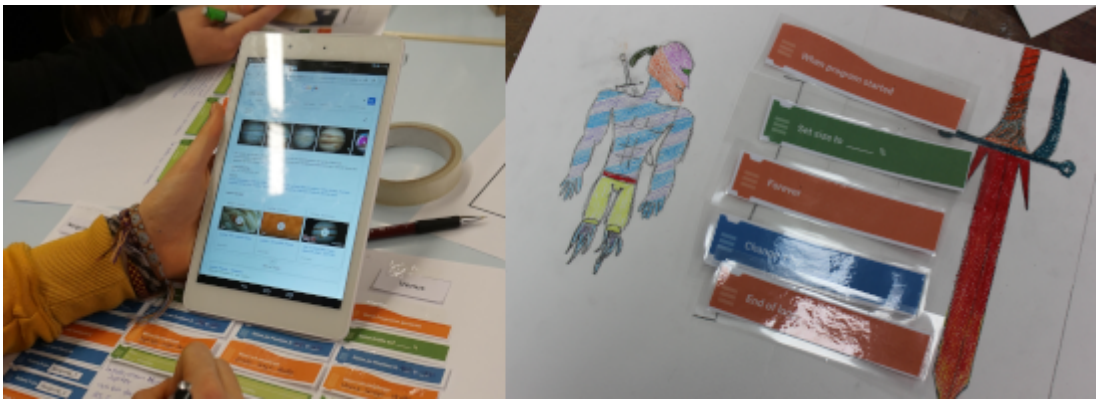

Figure 7: Students design a program to be run with Pocket Code

All in all, visual programming languages seem to provide an easier start and a more engaging experience for learners. The ease of use, simplicity, and desirability of new visual programming environments enables young people to imagine complex goals. A study which compared three classes that used either block-based (Scratch), text-based (Java), or hybrid blocks/text (Snap!/JavaScript) programming languages showed that students generally found block-based programming to be easier than the text-based environments (Weintrop and Wilensky 2015). Some researchers, however, argue that students are not fully convinced that a visual language can help them learn other programming languages (Lewis et al. 2014).

[8] https://snap.berkeley.edu/

[9] https://bjc.berkeley.edu/

[10] https://catrobat.org

Common features

The above short survey of programming languages for education shows they have some recurrent traits that link them to the themes discussed in the section "What does it mean to learn programming."

Personification The interpreter becomes a "persona", computation is then carried out through anthropomorphic (or, better, zoomorphic, since animals are very common) actions. This seems to contradict a famous piece of advice coming from no less than E. W. Dijkstra (Dijkstra 1985). Speaking of anthropomorphism in computer science, he noted: "The trouble with the metaphor is, firstly, that it invites you to identify yourself with the computational processes going on in system components and, secondly, that we see ourselves as existing in time. Consequently, the use of the metaphor forces one to what we call 'operational reasoning', that is reasoning in terms of the computational processes that could take place. From a methodological point of view, this is a well-identified and well-documented mistake: it induces a combinatorial explosion of the number of cases to consider and designs thus conceived are as a result full of bugs." The *reasoning in terms of the computational processes*, however, is what is probably needed for a novice in order to familiarize with the notional machine.

Visualization and tracking Computational processes that evolve in time are described by static texts: the mapping between the two is not trivial and it requires an understanding of the notional machine. Educational programming environments often try to make the mapping more explicit with some visualization of the ongoing process: the trace left by the LOGO turtle, or some other exposition of the changing state of the interpreter.

Appeal Engagement of learners is crucial: to this end, it is important to give learners powerful libraries and building blocks. It is not clear, however, how to properly balance amazing effects in order to avoid they become a major distraction: sometimes children may spend their (limited) time in changing the colors of the sprites, instead of trying to solve problems.

Learning to program in teams

Constructivist approaches often emphasize the importance of social context in which the learning happens (see e.g. (Vygotsky 1978)).
Working in developers teams requires new skills, especially because software products (even the ones in the reach of novices) are often tangled with many dependencies and division of labour is hard: it inevitably requires appropriate communication and coordination. Therefore, it is important that novice programmers learn to program in an "organized" way, discovering that as a group they are able to solve more challenging and open-ended problems, maybe with interdisciplinary contributions.
To this end, agile methodologies fit well with constructivist pedagogies involving learning in teams, and they are increasingly exploited in educational settings (see for example (Kastl, Kiesmüller, and Romeike 2016; Missiroli, Russo, and Ciancarini 2016)): indeed

- agile teams are typically small groups of 4–8 co-workers;
- agile values (Beck et al. 2001) (individuals and interactions over processes and tools; customer collaboration over contract negotiation; responding to change over following a plan; working software over comprehensive documentation) relate well with constructivist philosophies;
- agile teams are self-organizing, emphasize the need for reflecting regularly on

how to become more effective, and tune and adjust their behavior accordingly;
- agile techniques like pair programming, test driven development, iterative software development, continuous integration are very attractive for a learning context.

The iterative nature of agile methods is well exemplified by test-driven development, or TDD (Beck 2003). This technique reverses the order between code implementation and correctness test. Namely, the specification of the programming task at hand is actually provided with a test the *defines* correct behavior. The development cycle is then based on the iteration of the following procedure:

i. write a test known to fail according to the current stage of the implementation,
ii. perform the smallest code update which satisfies all tests, including the one introduced in the previous point, and
iii. optionally refactor the produced code.

TDD makes testing the engine driving the overall development process: one of the hardest-to-find contributions for facilitators in an active programming learning context is suggesting a good *next test*. This has the role of letting pupils aware that their belief at a broad level ("the program works") is false, thus an analogous belief at a smaller scale (for instance, "this function always returns the correct result") should be false, too. This amounts to the destruction of knowledge necessary to build new knowledge (aka a working program) in a constructivist setting. Moreover, refactoring corresponds to the constructivist re-organization of knowledge following the discovery of more viable solutions: most of the developing activities consist in *realizing* that a system which was thought to correctly work is actually not able to cope with a new test case. This applies of course also to the simplest tasks faced by students engaged in learning the basics of computer programming.
Once pupils are convinced that their implementation is flawed, the localization of the code lines to be reconsidered is the other pillar of an active learning setting. Again, a paramount contribution for a successful learning process should be provided by a facilitator suggesting suitable debugging techniques (e.g., proposing critical input values, suggesting points in the execution flow to be verified, or giving advice about variables to be tracked during the next run).

Conclusions

The literature on learning to program through a constructionist strategy has often focused on how to bring the abstract and formal nature of programming languages into the manipulation of more concrete (or even tangible) "objects" (Kay et al. 1997; Horn and Jacob 2007; Dann, Cooper, and Pausch 2008; Resnick et al. 2009; Hauswirth, Adamoli, and Azadmanesh 2017). Many proposals aim at overcoming the (initial) hurdles which textual rules of syntax may pose to children. Also, several environments have been designed in order to increase the appeal of programming by connecting this activity to real-world devices or providing fancy libraries. Instead, more work is probably needed to make educators and learners more aware of the so-called notional machine behind the programming language. Programming environments could be more explicit about the complex relationship between the code one writes and the actions that take place when the program is executed. Moreover, micro-patterns should be exploited in order to enhance problem solving skills of novice programmers, such that they become able to think about the solution of problems in the typical way that make the former suitable to automatic elaboration. Agile methodologies, now also common in professional settings, seem to fit well with constructionist learning. Besides the stress on

teamworking, particularly useful seems the agile emphasis on having running artifacts through all the development cycle and the common practice of driving development with explicit or even executable “definitions of done”.

References

Abelson H. & DiSessa A.A. (1986) *Turtle Geometry: The Computer as a Medium for Exploring Mathematics*. Artificial Intelligence Series. AAAI Press.

Abelson H., Dybvig R.K., Haynes C.T., Rozas G.J., Adams N.I., Friedman D.P., Kohlbecker E. et al. (1998) "Revised Report on the Algorithmic Language Scheme." *Higher-Order and Symbolic Computation* 11 (1): 7–105. https://doi.org/10.1023/A:1010051815785.

Abelson, H., Sussman G.J. & Sussman J. (1996) *Structure and Interpretation of Computer Programs*. Second. MIT press.

Allen E., Cartwright R. & Stoler R. (2002) "DrJava: A Lightweight Pedagogic Environment for Java." *SIGCSE Bull.* 34 (1). New York, NY, USA: ACM: 137–41. https://doi.org/10.1145/563517.563395.

Astrachan O. & Wallingford E. (1998) "Loop Patterns." In *Proceedings of the Fifth Pattern Languages of Programs Conference*.

Beck K. (2003) *Test-Driven Development: By Example*. Addison-Wesley Professional.

Beck K. & Andres C. (2004) *Extreme Programming Explained: Embrace Change*. Second. Addison-Wesley Professional.

Beck K., Beedle M., van Bennekum A., Cockburn A., Cunningham W., Fowler M., Grenning J. et al. (2001) "Manifesto for Agile Software Development." http://agilemanifesto.org/iso/en/manifesto.html.

Bell, T., Rosamond F. & Casey N. (2012) "Computer Science Unplugged and Related Projects in Math and Computer Science Popularization." In *The Multivariate Algorithmic Revolution and Beyond*, edited by Hans L. Bodlaender, Rod Downey, Fedor V. Fomin, and Dániel Marx, 398–456. Berlin, Heidelberg: Springer-Verlag. http://dl.acm.org/citation.cfm?id=2344236.2344256.

Bell T. & Vahrenhold J. (2018) CS Unplugged—How Is It Used, and Does It Work?. In: Böckenhauer HJ., Komm D., Unger W. (eds) Adventures Between Lower Bounds and Higher Altitudes. Lecture Notes in Computer Science, vol 11011. Springer, Cham.

Bell T. & Lodi M. (to appear) Constructing computational thinking without using computers. *Constructivist Foundations.*

Bellettini C., Lonati V., Malchiodi D., Monga M., Morpurgo A. & Torelli M. (2012) "Exploring the Processing of Formatted Texts by a Kynesthetic Approach." In *Proc. of the 7th Wipsce*, 143–44. WiPSCE '12. New York, NY, USA: ACM. https://doi.org/http://dx.doi.org/10.1145/2481449.2481484.

Bellettini C., Lonati V., Malchiodi D., Monga M., Morpurgo A. & Torelli M. (2013) "What You See Is What You Have in Mind: Constructing Mental Models for Formatted Text Processing." In *Proceedings of ISSEP2013*, 139–47. Commentarii Informaticae Didacticae 6. Universitätsverlag Potsdam. http://opus.kobv.de/ubp/volltexte/2013/6368/pdf/cid06.pdf.

Bellettini C., Lonati V., Malchiodi D., Monga M., Morpurgo A., Torelli M. & Zecca L. (2014) "Extracurricular Activities for Improving the Perception of Informatics in Secondary Schools." In *Proceedings of ISSEP2014*, edited by Yasemin Gülbahar and Erin c Karata s, 8730:161–72. Lecture Notes in Computer Science. Springer. https://doi.org/http://dx.doi.org/10.1007/978-3-319-09958-3_15.

Ben-Ari M. (2001) "Constructivism in Computer Science Education." *Journal of Computers in Mathematics and Science Teaching* 20 (1). Association for the Advancement of Computing in Education (AACE): 45–73.

Ben-Ari M. & Sajaniemi J. (2004) "Roles of Variables as Seen by Cs Educators." In *ACM Sigcse Bulletin*, 36:52–56. 3. ACM.

Berg J., (1999) "Patterns for Selection". In *Proceedings of the 4th European Conference on Pattern Languages of Programs* (EuroPLoP '1999): 305-326.

Berry M. & Kölling M. (2014) "The State of Play: A Notional Machine for Learning Programming." In *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education*, 21–26. ITiCSE '14. New York, NY, USA: ACM. https://doi.org/10.1145/2591708.2591721.

Brin D. (2016) "Why Johnny Can't Code." https://www.salon.com/2006/09/14/basic_2/.

Clancy M. (2004) "Misconceptions and Attitudes That Interfere with Learning to Program." In *Computer Science Education Research*, edited by Sally Fincher and Marian Petre, 85–100. Routledge.

Dann W.P., Cooper S. & Pausch R. (2008) *Learning to Program with Alice*. Prentice Hall Press.

Dijkstra E.W. (1985) "On Anthropomorphism in Science. EWD936." https://www.cs.utexas.edu/users/EWD/ewd09xx/EWD936.PDF.

Du Boulay, B. (1986). Some Difficulties of Learning to Program. Journal of Educational Computing Research, 2(1), 57–73. https://doi.org/10.2190/3LFX-9RRF-67T8-UVK9

W. Feurzeig, S. Papert, M. Bloom, R. Grant, and C. Solomon. 1970. Programming-languages as a conceptual

framework for teaching mathematics. *SIGCUE Outlook* 4, 2 (April 1970), 13-17. http://dx.doi.org/10.1145/965754.965757

Findler R.B., Bruce R., Clements J., Flanagan C., Flatt M., Krishnamurthi S., Steckler P. & Felleisen M. (2002) "DrScheme: A Programming Environment for Scheme." *Journal of Functional Programming* 12 (2). Cambridge University Press: 159–82.

Ford J.L. (2009) *Scratch programming for Teens*. In Computer Science Books.

Goldberg A. & Kay A. (1976) *Smalltalk-72 Instruction Manual*. Xerox.

Goode J., Chapman G. & Margolis J. (2012) "Beyond curriculum: the exploring computer science program." *In Magazine ACM Inroads*.

Hauswirth M., Adamoli A. & Azadmanesh M.R. (2017) "The Program Is the System: Introduction to Programming Without Abstraction." In *Proceedings of the 17th Koli Calling International Conference on Computing Education Research*, 138–42. Koli Calling '17. New York, NY, USA: ACM. https://doi.org/10.1145/3141880.3141894.

Horn M.S. & Jacob R.J.K. (2007) "Designing Tangible Programming Languages for Classroom Use." In *Proceedings of the 1st International Conference on Tangible and Embedded Interaction*, 159–62. TEI '07. New York, NY, USA: ACM. https://doi.org/10.1145/1226969.1227003.

Kahn K. (2017) "A half-century perspective on Computational Thinking." *In Technologias, Sociedade E Conhecimento*.

Kastl P., Kiesmüller U. & Romeike R. (2016) "Starting Out with Projects: Experiences with Agile Software Development in High Schools." In *Proceedings of the 11th Workshop in Primary and Secondary Computing Education*, 60–65. WiPSCE '16. New York, NY, USA: ACM. https://doi.org/10.1145/2978249.2978257.

Kay A.C. (1993) "The Early History of Smalltalk." *SIGPLAN Not.* 28 (3). New York, NY, USA: ACM: 69–95. https://doi.org/10.1145/155360.155364.

Kay A., Rose K., Ingalls D., Kaehle T., Maloney J. & Wallace S. (1997) "Etoys & SimStories." *Walt Disney Imagineering*.

Kurtz T.E. (1978) "BASIC." *SIGPLAN Not.* 13 (8). New York, NY, USA: ACM: 103–18. https://doi.org/10.1145/960118.808376.

Colleen L., Esper E., Bhattacharyya V., Fa-Kaji N., Dominguez N. & Schlesinger A. (2014) "Children's perceptions of what counts as a programming language." *In Journal of Computing Sciences in Colleges*.

Lonati V., Monga M., Morpurgo A. & Torelli M. (2011) "What's the Fun in Informatics? Working to Capture Children and Teachers into the Pleasure of Computing." In *Proceedings of Issep2011*, edited by I. Kalaš and R.T. Mittermeir, 7013:213–24. Lecture Notes in Computer Science. Springer-Verlag. https://doi.org/http://dx.doi.org/10.1007/978-3-642-24722-4_19.

Maloney J., Resnick M., Rusk N., Silverman B. & Eastmond E. (2010). "The Scratch Programming Language and Environment." *Trans. Comput. Educ.* 10 (4). New York, NY, USA: ACM: 16:1–16:15. https://doi.org/10.1145/1868358.1868363.

Meerbaum-Salant O., Armoni M. & Ben-Ari M. (2010) "Learning computer science concepts with scratch." *In Proceedings of the Sixth International Workshop on Computing Education Research*, 69–76.

Missiroli M., Russo D. & Ciancarini P. (2016) "Learning Agile Software Development in High School: An Investigation." In *Proceedings of the 38th International Conference on Software Engineering Companion*, 293–302. ICSE '16. New York, NY, USA: ACM. https://doi.org/10.1145/2889160.2889180.

Piaget, J. (1973). To understand is to invent: The future of education. Penguin Books.

Papert S. (1980) *Mindstorms: Children, Computers, and Powerful Ideas*. New York, NY, USA: Basic Books, Inc.

Proulx V.K. (2000) "Programming Patterns and Design Patterns in the Introductory Computer Science Course." In *ACM Sigcse Bulletin*, 32:80–84. 1. ACM.

Qian Y. & Lehman J. (2017) "Students' Misconceptions and Other Difficulties in Introductory Programming: A Literature Review." *ACM Trans. Comput. Educ.* 18 (1). New York, NY, USA: ACM: 1:1–1:24. https://doi.org/10.1145/3077618.

Resnick M. (1996) "Distributed Constructionism." In *Proceedings of the 1996 International Conference on Learning Sciences*, 280–84. ICLS '96. Evanston, Illinois: International Society of the Learning Sciences. http://dl.acm.org/citation.cfm?id=1161135.1161173.

Resnick M. (2007) "All I Really Need to Know (About Creative Thinking) I Learned (by Studying How Children Learn) in Kindergarten." In *Proceedings of the 6th Acm Sigchi Conference on Creativity &Amp; Cognition*, 1–6. C&C '07. New York, NY, USA: ACM. https://doi.org/10.1145/1254960.1254961.

Resnick M. (2017) *Lifelong Kindergarten: Cultivating Creativity Through Projects, Passion, Peers, and Play*.

MIT Press.

Resnick M., Maloney J., Monroy-Hernández A., Rusk N., Eastmond E., Brennan K., Millner A. et al. (2009) "Scratch: Programming for All." *Commun. ACM* 52 (11). New York, NY, USA: ACM: 60–67. https://doi.org/10.1145/1592761.1592779.

Rodger S., Hayes J., Lezin G, Qin H., Nelson D., Tucker R., Lopez M., Cooper S., Dann W., Slater D. (2009). Engaging middle school teachers and students with alice in a diverse set of subjects. *SIGCSE Bull.* 41, 1 (March 2009), 271-275. https://doi.org/10.1145/1539024.1508967

Sajaniemi J. (2002) "An Empirical Analysis of Roles of Variables in Novice-Level Procedural Programs." In *Human Centric Computing Languages and Environments, 2002. Proceedings. IEEE 2002 Symposia on*, 37–39. IEEE.

Sirkiä T. (2012) "Recognizing Programming Misconceptions: An Analysis of the Data Collected from the Uuhistle Program Simulation Tool." Master's thesis, Department of Computer Science; Engineering, Aalto University.

Slany W. (2014) "Tinkering with Pocket Code, a Scratch-like programming app for your smartphone." *In Proceedings of Constructionism 2014*.

Sorva J. (2013) "Notional Machines and Introductory Programming Education." *Trans. Comput. Educ.* 13 (2). New York, NY, USA: ACM: 8:1–8:31. https://doi.org/10.1145/2483710.2483713.

Taub R., Armoni M. & Ben-Ari M. (2012) "CS Unplugged and Middle-School Students' Views, Attitudes, and Intentions Regarding CS." *TOCE* 12 (2): 8. https://doi.org/10.1145/2160547.2160551.

Tumlin N. (2017) "Teacher Configurable Coding Challenges for Block Languages." *In Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*.

Vygotsky, L. (1978). Mind in Society. London: Harvard University Press.

Weintrop D. & Uri W. (2015) "To block or not to block, that is the question: students' perceptions of blocks-based programming." *IDC '15 Proceedings of the 14th International Conference on Interaction Design and Children*.

Wirth N. (1993) "Recollections About the Development of Pascal." *SIGPLAN Not.* 28 (3). New York, NY, USA: ACM: 333–42. https://doi.org/10.1145/155360.155378.

The authors

Michael Lodi is a PhD student in Computer Science, Department of Computer Science and Engineering, University of Bologna, Italy. He also received Bs, Ms and High school teaching licence in CS from the same University.
He works on computer science education, with a particular focus on teacher training about computational thinking and epistemological aspects of Computer Science as a discipline. In particular, he studies "Computer Science Growth Mindset".
He published some papers in international conferences on computer science education, and a book in Italian for primary school teachers.
He is actively involved in nation-wide initiatives to introduce CS in Italian K-12 curriculum. https://lodi.ml

Dario Malchiodi (http://malchiodi.di.unimi.it) is an associate professor at Università degli Studi di Milano (Department of Computer Science), where he teaches "Statistics and data analysis" and "Big scale analytics". His research activities are focused on the one hand on the treatment of uncertainty in machine learning problems, and on the other

one on the development of teaching methodologies for primary and secondary education. He published around one hundred papers and he participated in the activities of ten research projects and research groups, at national and international level. He is co-founder of the ALaDDIn working group (http://aladdin.unimi.it), involved in several activities focused on the popularization of computer science, including the training of secondary school teachers and a radio broadcast on informatics culture.

Mattia Monga is an Associate Professor at Università degli Studi di Milano (Department of Computer Science). His research interests are mainly in the field of software engineering, system security, and computer science education. Since he believes it is urgent to change the common misconception of informatics as the mere use of information technologies, he founded together with Carlo Bellettini, Violetta Lonati, Dario Malchiodi, and Anna Morpurgo a group working to spread informatics as a science among the general public (https://aladdin.unimi.it/). Moreover, he is the National Bebras Organizer for Italy.

Anna Morpurgo is an Assistant Professor at the Department of Computer Science, Università degli Studi di Milano, Italy. Her current research interests are mainly in computer science education. She is co-founder of the ALaDDIn working group (http://aladdin.unimi.it), active in the popularization of informatics as a science in school and among the general public. She is involved in the training of secondary school teachers and is part of the team organizing the Bebras challenge in Italy.

Bernadette Spieler has a PhD in Engineering Sciences. She is a University Assistant at Graz University of Technology, Department for Software Technology. Her work is focused on how to encourage female teenagers with playful coding activities with the Pocket Code app or in “Girls Coding Weeks”. Moreover, her recent work is related to gender, game based/mobile learning and constructionist gaming. Through the nonprofit university project Catrobat, Mrs. Spieler promotes computational thinking skills in a fun and engaging way among children, teenagers and teachers on a worldwide scale. https://bernadette-spieler.com