Ca' Foscari
University
of Venice

Master's Degree programme

in Computer Science

Second Cycle (D.M. 270/2004)

Final Thesis

Didactics of Computational Thinking Addressed to Non-Computer Science Learners

Supervisor
Dott. Marta Simeoni, Dott. Teresa Scantamburlo

Graduand
Prisca Primavera Piccoli
Matriculation Number 832672

Academic Year
2016 / 2017

Table of Contents

Introduction

Despite the growing pervasiveness of computing and technology in our everyday lives, now influencing almost every disciplinary and professional field, many companies from the IT sector are reporting a crippling shortage of qualified professionals. In addition, we are witnessing a declining phase of students' interest and enrollment in Computer Science courses. In response to this, Computer Science education in primary and secondary schools as well as universities is currently going through a remarkable transformation.

Computational Thinking (CT), the term in use to refer to the key ideas and concepts of the disciplinary area of Computer Science (CS), has been proposed as a universally applicable way of thinking whose benefits can be experienced by practitioners from many sectors, not just computer scientists. Indeed, the focus on computational thinking does not simply allow students to become familiar with the use of software. Conversely, it goes beyond computer literacy, and can be seen as a way to introduce students to those concepts and cognitive approaches that are needed to Computer Science, in order to motivate their interest for this discipline. Moreover, starting to face computational thinking from early childhood has also been proved to reduce the difficulties that undergraduates are known to experience at the beginning of university Computer Science courses.

In the last few years, Computer Science education addressed to school-aged children and non-CS university students has been gaining increasing attention from researchers, professionals, and policymakers in the field of education. As a result, there has been a notable increase in the amount of both academic and gray literature on computational thinking, which is also being more and more often mentioned, explicitly or implicitly, in policy-related documents.

A number of prominent institutions all over the world, such as the Royal Society (UK), the Académie des Sciences (France) and the Association for Computing Machinery (USA) have intervened in favor of the introduction of computational skills in the context of compulsory education. Industry also supports the idea of giving Computer Science the status of a discipline on its own. Furthermore, international debate has put in evidence the importance of Computer Science

studies not just as a content area that is crucial in nowadays social context, but also for the potential to develop general cognitive skills and digital competences, especially as far as coding is concerned.

In consideration of this, the purpose of the present work is to provide an overview of the current debate over the role of computational thinking inside primary and secondary education, by analyzing some of the most recent didactic proposals, and to suggest possible directions for future enquiries.

The thesis is organized as follows.

The first Chapter introduces the notion of computational thinking and illustrate the different cognitive approaches that can be included within this thinking method. Subsequently, a quick overview of the existing literature on this topic is also provided.

The second Chapter presents some of the most significant initiatives that have been started during the last decade in the United States and in Europe (particularly with respect to the United Kingdom, Ireland, France and Italy), in order to allow children to approach coding right from their school age.

The third Chapter illustrates the features of some of the most common didactic tools used by educational institutions to introduce primary school children and secondary school students to the basics of coding. We will particularly focus on some popular block programming languages, such as Scratch, Snap!, Stencyl and the Blockly library, highlighting similarities and differences among them.

The fourth Chapter analyzes some recent didactic proposals that rose in response to the call to action from the academic community and the world of industry in favor of the introduction of computational thinking inside compulsory education, and presents the most significant results from research in this field.

The fifth and last Chapter concludes the work by presenting a personal proposal for an introductory coding course addressed to non-Computer Science students.

Chapter 1

Computational Thinking

This chapter will introduce the concept of computational thinking and will provide an overview of the available literature on the topic.

1.1 Introduction to computational thinking

Computational Thinking (**CT**) can be defined as a problem solving process characterized by a set of peculiar features and dispositions. CT plays a major role in the creation of computer applications, nonetheless it can also be useful as a support to problem solving in a wide range of disciplines, including mathematics, science and the humanities [1]. Students learning CT are generally more inclined to notice similarities between different school subjects, as well as between what they study and what they experience in their everyday life [46]. In fact, CT can also be applied to answer usual questions such as, for example: how can we map the entire human DNA sequence? Is it really possible that William Shakespeare wrote all of the works that are attributed to him? Is it possible to create a computer program that is able to create its own novel music composition?

Computational Thinking basically consists of taking a problem apart and figuring out a possible solution by exploiting our knowledge about computation. Four major facets of CT can be identified:

1. **Decomposition**: breaking down problems, processes or data into smaller and more easily manageable components;
2. **Pattern Generalization** or **Abstraction**: identifying the general mechanisms underlying the generation of these patterns;

3. **Pattern Recognition**: discovering regularities, trends and patterns inside these components with the purpose of making predictions;
4. **Algorithm Design**: defining the step-by-step instructions that allow to solve the problem currently studied as well as similar problems [59].

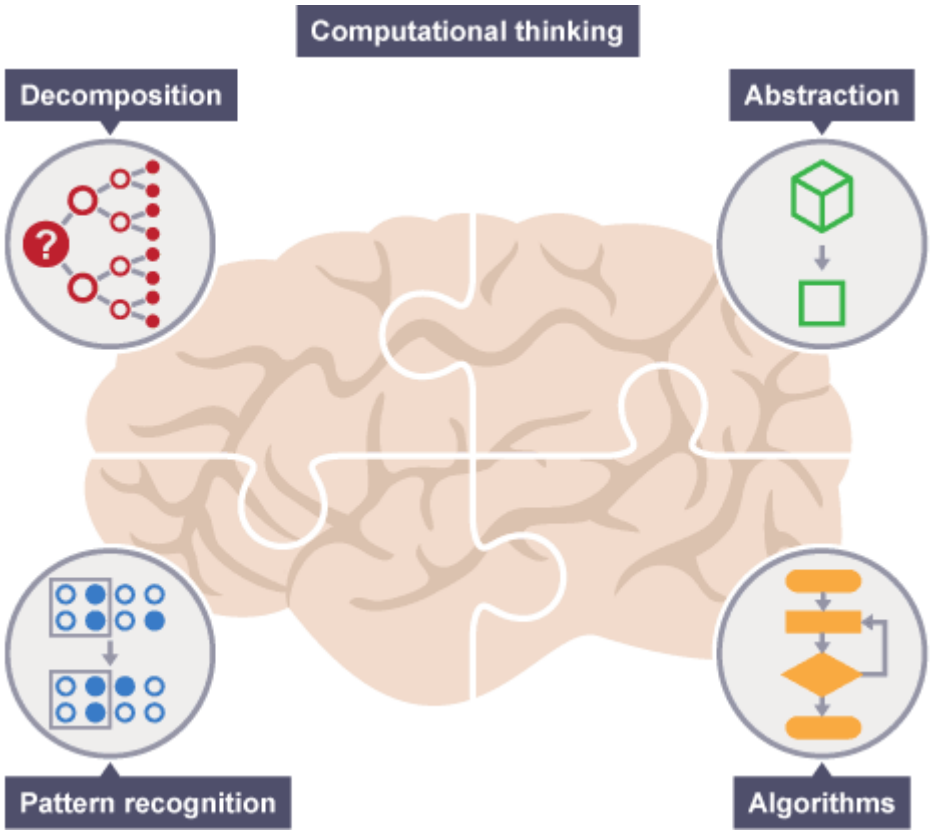

Figure 1.1. The four cornerstones of computational thinking

In the last few years, more and more companies and educational institutions have realized that the ability to think computationally can be effectively exploited in many disciplines. In fact, whenever a difficult question arises that can be easily solved by a computer, or insights are discovered through the analysis of large data sets, computational thinking is actually being applied.

CT has supported the development of fields such as computational biology and computational chemistry, and it is also thanks to CT that we can now use new techniques for studying humanistic disciplines, such as literature, social studies, and the arts. If we just start looking around us, we can easily realize that CT is everywhere [63].

1.2 Computational thinking and computer science

The phrase *computational thinking* is commonly used as a synonym for computer science, but these two concepts are instead quite distinct. Computer science is an academic discipline that encompasses the study of computation and applies it by means of computers, whereas computational thinking is a way of tackling problems, through big picture processes and abstract thinking approaches, which can be used regardless of whether one's object of study is computer science or something else. In other words, thinking computationally does neither mean coding nor thinking like a computer, as, of course, computers cannot think. If programming can be described as telling a computer what it must do and how to do it, computational thinking is what allows us to define exactly what we should tell the computer to do.

For example, if we agree to meet someone at a certain place we have never known previously, we would probably need to plan our path. In particular, we should analyze the possible alternatives and evaluate which one is best accordingly to our needs (this could be the fastest route, the shortest one, or the one passing by a given point). Once we have established our itinerary, we should reach the preset location by following the step-by-step directions. In such a situation, the planning phase would represent computational thinking, while following the directions would represent coding.

Several studies carried out by leading computer scientists such as Mitchel Resnick [42] (Massachusetts Institute of Technology) and Jeannette Wing [52] (Carnegie Mellon University) suggested that CT increases students' confidence in dealing with ambiguous, complicated or unsolved problems. As a consequence, they suggested that all students should learn CT, regardless of their chosen study track or age group. Indeed, being able to face a problem, divide it up into many differently sized parts, work on each one separately, and eventually join everything together with the goal of solving the initial problem, actually constitute skills that could benefit professionals of every field.

In what follows, we will analyze in more detail the different aspects constituting CT.

1.3 Decomposition

Decomposition consists of splitting a complex problem or system into a number of elementary components that are easier to manage and to understand. Since these elementary components are simpler to work with, they can be designed, analyzed or resolved individually.

It is easily deducible that a problem can prove to be much harder to solve if it is not decomposed, as facing several difficulties at the same time can be much more complicated than

breaking a problem down into a number of easier sub-problems and trying to solve each of them singularly. Moreover, if smaller problems are considered, this means that each of these can be analyzed in a more detailed way. Similarly, the mode of operation of a complex system can be more easily understood by means of decomposition.

For example, when a programmer is required to create an application, he needs to decompose his task into the following sub-problems:

- What kind of application he has to realize;
- What is the purpose of the application;
- What should be the sub-functions composing the application;
- What his application should look like;
- What is the target audience of his application;
- What should be the graphic appearance of the application;
- Whether and what kind of audio should be included;
- What programming language he will use to build his application;
- What should be the user interface;
- How the application will be tested;
- Where the application will be sold.
- …

In such a way, the complex problem of creating an application has been divided up into a set of simpler problems that are now easier to manage. Moreover, this division makes it possible for different experts to collaborate on the same project by devoting themselves to a specific aspect of the problem. For example, a developer might deal with the programming part, while a designer might take care of the graphics.

1.4 Abstraction

Pattern generalization, or abstraction, consists of excluding from our analysis those elements that are unnecessary, with the aim of concentrating on those that are actually needed.

As we have previously explained, in computational thinking, after decomposing a problem, we need to search for common patterns among and within the elementary problems that constitute the overall complex problem. Abstraction can be defined as the process of ignoring the superfluous characteristics and the specific details of the patterns we have found, in order to concentrate on the

ones that are necessary to our purposes. Thanks to this filtering process, we can obtain a simplified representation, or *model*, of the system we are studying.

When speaking about pattern recognition, we dealt with the problem of drawing a set of dogs, and noticed that all dogs share some common physical features, such as eyes, tail, and fur. Besides that, they all like meat and can bark. Of course, every single dog has its own specific characteristics, such as brown eyes, a thick fur, or a long tail. In addition, its favorite food might be pork, and its barking might be particularly high pitched. These characteristics are commonly referred to as specifics.

Nevertheless, if our task is that of drawing a dog, we do not need to know what its barking sounds like or what kind of meat it prefers, because these characteristics are completely irrelevant for our purpose and thus can be ignored. The only relevant characteristics we need to be aware of are those related to the physicalness of a basic dog.

By means of abstraction, we are able to derive a more general idea of our problem and of its possible solutions. This process, in fact, consists of excluding from our analysis all the irrelevant details of our system, and the patterns that are not needed to perform our task, which allows us to create what is known as a model.

A model can thus be defined as a general representation of the problem we want to solve. For example, a model aimed at representing a generic dog should be adaptable to any dog, and, based on this model, we should be able to represent any dog by exploiting our knowledge with respect to the features that are shared by all dogs. Defining a model for our problem, in fact, allows us to design a precise algorithm for solving it.

1.5 Pattern recognition

Once a problem or system has been decomposed, it often happens to discover common features among the smaller problems we are analyzing. These features are usually called patterns. Pattern recognition, in fact, is a research branch of machine learning focusing on the identification of regularities among data [4]. From an algorithmic viewpoint, it can be defined as the set of actions we perform in order to formulate, select, modify and adjust our reference concepts so that we can unmask a known *form* inside an unknown object, where by form we mean a low-entropy set of *well-structured* or *well-organized* data points. In accordance with this point of view, the theoretical physicist Satosi Watanabe also referred to the process of pattern recognition as *conceptual morphogenesis* [47]. Moreover, the expression also identifies a field of study related to artificial intelligence.

More pragmatically, pattern recognition consists of examining a set of elementary problems obtained through decomposition and discovering the similarities or common patterns among them which can help us to resolve the global problem more efficiently.

Indeed, patterns are common characteristics shared by a set of elements. For example, if we wanted to draw a generic dog, we should include the common features of dogs, such as ears, eyes, nose, paws, tail and fur. If instead we wanted to draw a particular dog, all we need to do is to follow this pattern modifying the specifics accordingly. Indeed, one dog may have brown eyes, a short tail and black fur, while another one may have blue eyes, a long tail and spotted fur.

In the common language, the term *pattern* can be commonly substituted, depending on the context, with terms such as *design*, *model*, *scheme*, *recurring scheme*, *recurring structure*, and the expression *pattern recognition* is also sometimes used as a synonym for *machine learning*, which is probably due to the significant relationship between the activity of recognizing patterns and the subsequent modeling [4]. Indeed, patterns correspond to a specific category of models, that the English philosopher of science Mary B. Hesse called *analogical models* [21].

In particular, she distinguished between two main types of analogies, namely *material analogies* and *formal analogies*.

Material analogies are represented by those cases in which two objects share a set of properties, or there is a relevant similarity between their properties. For example, we could state that dogs and cats share a set of features as both of them have two ears, four paws and one tail. A more liberal reasoning may lead us to state that light and sound share the features of reflection, brightness and color, which respectively correspond to the phenomena of echo, loudness and pitch in the case of sound.

On the contrary, formal analogies exist when a given analogue model does not share a real set of features with its target, but the same relational structure, or pattern, indeed. In mathematical terms, two elements are related by formal analogy if they can be described by the same set of equations.

The ability to recognize patterns can turn out to be extremely useful in making our task easier. Indeed, problems that share common patterns prove to be simpler to solve, because a similar solution can be applied to all cases where a similar problem exists. Thus, the larger is the number of patterns we can discover, the more simply and fast we will find a solution to our overall problem.

A common way to provide a pattern is to identify a number of features characterizing the objects of a specific class. For example, in the case of dogs, if we wanted to draw a set of specimens, identifying a pattern followed by dogs in general would be an effective help for our purpose. In fact, once we know that all dogs share ears, eyes, nose, paws, tail and fur, we would not

need to analyze in detail how dogs look like every time we start drawing a new one, which would make our task much easier and quicker.

Without having any pattern to conform to, we could still succeed in drawing our dogs, but every single dog would take us far longer to draw, which would be a very inefficient way of working out the dog-drawing problem. Even more importantly, without discovering patterns we might not realize that all dogs share the features we have listed, which could make them not even recognizable as dogs. In this case, we would be providing an incorrect solution to our problem.

Patterns can be detected **among different problems** or **within single problems**. Of course, we should look for both kinds of patterns.

In order to find patterns among different problems we should search for features that are equal or very similar in each problem, which corresponds to finding common solutions to similar problems. Trying to identify the features that are common to a set of dogs is an example of this kind of approach.

Patterns existing within single problems, instead, result from the decomposition of the initial problem. For example, if our dog representation should be created by means of a set of small monochrome squares (such as pixels) we should consider the sub-problems of determining, for each body part of our dog, the number of pixels to be allocated, the distribution of these pixels and their color. Once we know that each body part is identified by these three features, we will also be able to apply this pattern to all the remaining body parts.

1.6 Algorithm design

In computer science, an algorithm consists of a plan, a self-contained set of step-by-step operations to be performed in order to solve a problem [29]. An algorithm can be considered as such if all the instructions that constitute it are identified in an unambiguous way and their execution order is scheduled in advance. When a developer is asked to create a computer program, he will generally need to start from an algorithm, which can be represented as a flowchart or written in pseudocode.

Of course, computers cannot and do not think autonomously, and their computational capacity is simply dependent of the algorithms that are run on them. If a computer is provided with a poor algorithm, the result obtained will be inevitably poor as well. Hence it originates the common saying: *Garbage in, garbage out*. Algorithms can be effectively used in a number of different fields, including mathematics, data processing and automation.

When designing an algorithm, it is fundamental to make sure that the solution provided will be correct[1] or, at least, a sufficiently good approximation of the exact solution. Applying decomposition will allow us to divide up the problem into a set of elementary sub-problems, which will be faced in a suitable order to get to the solution of the overall problem. Of course, this order will be specified by the algorithm, which must be unambiguous, and must include a starting point, generally containing the initialization of the relevant variables, a finishing point, consisting of the return of the required results, and a central body, including a number of well defined instructions to follow.

Algorithms are usually represented by using one of these two methods: **pseudocode** and **flowcharts**.

In practice, in computer science, algorithms are referred to as programs and are developed by means of programming languages. Each language has its own specific syntax, that every programmer must respect in order to make his algorithm correctly interpretable by the machine. Differently, pseudocode simply consists of listing a number of instructions, and does not envisage any specific syntax. Therefore, it cannot be considered a programming language.

Nevertheless, algorithms written in pseudocode look similar to real programs, because each line of the algorithm contains a single instruction. Moreover, it is common use to indicate operations in uppercase, variables in lowercase and strings in sentence case. Finally, the INPUT and OUTPUT keywords are used to indicate respectively what the user is expected to type and what is printed on the screen.

Going back to the dog drawing problem, let us assume that we have successfully managed in the task of drawing a diverse set of dogs, and we are now asked to compute their average height. Of course, as a first thing, all the relevant height measures taken from the drawings should be transformed into the corresponding real measures. Secondly, we could apply an algorithm such as the one provided in the following example, that is a simple algorithm written in pseudocode that computes the average of a set of values given as input.

START
INPUT a list of **N values**
INITIALIZE the **sum** equal to zero
REPEAT for **i** ranging from 1 to **N**, INCREMENT by 1 at a time
    COMPUTE the **sum** by adding the **i**-th element of the list of **values** to the previous **sum**

[1] An algorithm is said to be *correct* if, for every input instance, it halts with the correct output, or, in other words, it *solves* the computational problem [29].

END of the for loop

COMPUTE the **average** as the final **sum** divided by **N**

OUTPUT the **average**

END

Another way of representing algorithms is by means of flowcharts, special diagrams that indicate different kinds of instructions by using specific symbols. There is no standard rule with respect to the level of detail that should be provided by a flowchart. In some cases you may need to break the algorithm down into a huge number of steps, in order to provide a detailed explanation of what the procedure is actually doing at any given time. In other cases, for clarity purposes, you may need to give a more simplified representation, by bringing back together a set of operations into one single step.

The previous algorithm written in pseudocode can be also represented by using a flowchart, as illustrated in Figure 1.2.

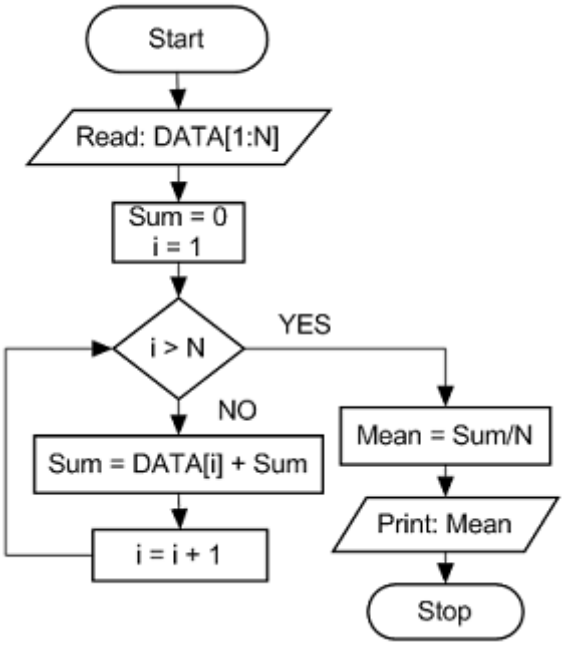

Figure 1.2. Flowchart representation of an algorithm computing the average of a set of values

1.7 Solution evaluation

The process of solution evaluation basically consists of verifying whether the conceived solution actually works with respect to the problem it has been designed for, and thinking about possible improvements.

More specifically, after designing an algorithm, a programmer should make sure that it satisfies the following requirements:

- full **decomposition** – each of its elementary components must be correctly identified and treated;
- **completeness** – it must be able to face all possible declinations of the problem;
- **efficiency** – it must solve the problem with the least possible time and space complexity, in order to better exploit the available system resources;
- **requirements meeting** – it must meet all the established design criteria.

Once the designed algorithm has been proved to fulfill all these requirements, the difficulties arising when writing the corresponding program are likely to be as few as possible. Otherwise, a faulty or incomplete solution algorithm may lead to remarkable implementation difficulties or, even worse, to writing a program that does not solve the assigned problem correctly. The evaluation process is then necessary to the programmer in order to make sure that the algorithm conceived actually meets the established design criteria, is able to solve the problem it was designed for, and is sufficiently efficient.

For example, a problem that is not completely decomposed would result in ineffective, and consequently wrong, solutions, because of the impossibility to tackle with the elementary phases of the problem resolution. If we go back to the dog-drawing problem, and we were asked to draw a group of dogs while playing in a garden, at a certain point the algorithm would state that we need to draw a garden. Nevertheless, if this phase of the algorithm was not further specified in its components, such as the color and the size of the garden, we would risk to draw something that is not clearly recognizable as a garden.

Incomplete solutions arise when one or more aspects of the problem are left out. In our example, this would be the case if the algorithm did not even mention that a garden needs to be drawn. Of course, an incomplete solution algorithm would make our chances to obtain an effective representation quite slim.

Inefficient solutions consist of programs that can solve a given problem, but are unnecessarily long and complicated. With respect to our example, any solution algorithm, at a certain point, should tell us to collect the set of colors that are necessary to represent a given dog. Nevertheless, it might either tell us to fetch all the needed colors in one shot, and then put them together in our palette to paint the dog, or it could give us a single fetching instruction for each color. Of course, in the first case, the procedure would be quicker, simpler and more efficient.

Eventually, solutions need to be evaluated in terms of their accordance with the original design criteria. This means that a solution should not drift too much apart from the original requirements,

that it should actually solve a wide range of instances of the problem and should be user-friendly. With reference to the dog-drawing problem, if we were asked to represent a group of German Shepherds, the algorithm should allow us to make our dogs recognizable as such, and not simply lead us to represent dogs of generic and stereotyped breed type.

In closing, solutions may be evaluated in different ways. In particular, to make sure of the correctness of a given solution, it is important to wonder whether it fulfills the following requirements.

- Does the solution allow you to fully understand how to solve the problem? If there are some unclear passages of the procedure, that is, there is something you still do not know how to do, it means that the problem has not been completely decomposed.
- Does the solution cover all aspects of the problem? If one or more steps are missing, then the solution is incomplete.
- Does the solution include repeated instructions?[2] If this is the case, reduce as much as possible the number of repetitions to make the solution more efficient.

After verifying that the solution fulfills the described requirements, the algorithm must be tested. One of the most common ways of testing it consists of what is referred to as a *dry run*, that is, working through the solution by following the instructions it provides. Of course, if a wrong result is obtained at the end of the procedure, then the algorithm is incorrect and needs to be fixed.

1.8 The literature on computational thinking: Papert, Wing, Resnick

The term *computational thinking* was first introduced in 1980 by Seymour Papert, a South African-born American mathematician, computer scientist, and educator, who devoted a large part of his career to teaching and research activities at Boston MIT. If we wanted to evaluate scientists' legacy based on how many people their projects touched, Papert would be without doubt one of the most influential scientists of the twentieth century. This is particularly due to the release of *Logo* in 1968, a revolutionary programming language developed by Papert in collaboration with the MIT researchers Cynthia Salomon, Daniel Bobrow, and Wally Feurzeig. Indeed, it was the first language expressly designed to be used by children. The novelty it constituted is even clearer if we remember that, in the 1960s, computers were generally as big as entire rooms and highly complex to manage.

The central aspect of Papert's vision was, in fact, that *children should be programming the computer, rather than being programmed by it* [6]. What is more, through his researches he arrived

[2] This aspect is related to the concept of *modularity*, that is the degree to which a system's components are made up of relatively independent parts which can be combined [84].

to discover that children can understand and memorize things more easily if they can see a tangible result of the applications they have crafted. This means that allowing them to program the behavior of physical objects makes the exercise much more useful than simply asking them to control what happens on a screen. This is why he created a robot turtle that could be commanded by means of Logo instructions, in order to draw geometrical shapes.

Papert also was one of the first interaction designers who especially devoted their efforts to children's education to computing. Being conscious of the different modalities in which children and adults think and learn, he designed several toolkits and programming environments that are much better suited to children's cognitive style rather than force-fed knowledge, setting the tone for future research in the field of education. In the last few years, the path started by Papert has been resumed by several renowned scholars.

One of these is Jeannette Wing, another strong promoter of computing education who is currently Corporate Vice President of Microsoft Research and has served till 2013 as the President's Professor of Computer Science at Carnegie Mellon University, Pittsburgh (PA). According to her view, *computational thinking will influence everyone in every field of endeavor* [54], from science to the humanities.

Indeed, the recent debate over CT can be seen as an effect of computing's pervasiveness, particularly for what concerns some branches of it. For example, Wing says, CT is transforming the field of statistics, where the development of artificial intelligence, combined with the use of Bayesian probabilistic models, has made possible the identification of regular patterns and anomalies inside voluminous datasets describing highly diverse quantities, such as astronomical data, credit card purchases and grocery store receipts. CT is modifying the field of biology, first thanks to the shotgun sequencing algorithm, which has notably shortened the time needed to sequence the entire human genome, and then with the abstractions representing natural processes such as cell cycle and protein folding. CT is changing economics, where a new branch of computational microeconomics is emerging, including applications for online auctions, advertisement placement, reputation services, and even for retrieving organ and tissue donors.

Furthermore, Wing envisions that CT will get even *deeper* in the future, thanks to the development of more ingenious and sophisticated abstractions, so enabling researchers to analyze and model systems of greater orders of magnitude than the ones they can handle nowadays.

Coherently with her view that CT will be used everywhere, Wing deduces that *it will touch everyone directly or indirectly* [54]. Nevertheless, as such, it constitutes an educational challenge, as educators should reflect about how and when people could learn this thinking approach more effectively, and how and when it should be taught. Considering that CT is already being used in

many disciplinary fields, thus influencing several university study tracks, and that many universities have included it in their undergraduate curricula, she suggests that students should be educated in CT starting from their early childhood, in order to allow them to acquire a solid understanding of this thinking approach.

Important contributions to the diffusion of computing education are also owed to Mitchel Resnick, currently serving as Director of the Lifelong Kindergarten group at the MIT Media Lab[3], and mainly known for being the inventor of Scratch, a block programming language aimed at making it easier for children to create their own interactive applications, videogames and animated stories.

According to Resnick, the nickname of *digital natives*, commonly attributed to kids from the younger generations, is quite improper. In fact, being very familiar with browsing the web, texting and playing videogames does not necessarily make them *fluent* with the technology around them, as being fluent cannot simply correspond to a passive usage of these tools. Similarly to what happens when learning a language, we can say that a person is fluent when he/she is not only able to *read*, but also to *write* something of his/her own. With respect to technology, fluency cannot be disjointed from the ability to write something with the language of technology, in order to express one's own ideas thanks to the tools it provides. This is why he developed Scratch, to help young people become fluent with new technologies, which concretely means being able to create their own programs, or code [43].

[3] The MIT Media Lab is an interdisciplinary research laboratory at the Massachusetts Institute of Technology devoted to projects at the convergence of technology, multimedia, sciences, art and design [86].

Chapter 2

Projects and Initiatives for IT Education

In the course of this chapter, we will provide an overview of the most significant projects and initiatives that are currently being carried out in the United States and in Europe in order to allow children to approach coding right from their school age.

2.1 Introduction

In the last few years, we are witnessing an enormous growth of the initiatives for computing education, strictly related to the increasing demand for skilled programmers inside large tech companies. Indeed, it is presumably true that a large number of students will need some programming knowledge in their future. Nevertheless, not all of them will use the same tools for the same purposes, and this is why there exist different viewpoints with respect to the utility of a widespread computing education [16].

In fact, many programmers do not agree with what Steve Jobs said, according to which *everybody should learn how to program a computer, because it teaches you how to think* (Steve Jobs interview from Triumph of the Nerds, 1995). Those who are against a programming education targeting a large audience think that such an initiative might give rise to a set of problems, including the following.

Primarily, people who were not trained as professional software developers are generally unable to create useful and usable applications, but they might nonetheless be interested in applying

for programming jobs, in most cases overestimating their qualifications. Secondly, even if they had reached a good skill level in using some specific technologies, they should consider that, sooner or later, every tool they have learned will go out of style, and they will need a more flexible understanding of programming methods and hardware itself, in order for their competences to be valuable in the IT field. Thirdly, and maybe most importantly, our society is based on the division of labor, and most technology users will never need to understand how the tools they use actually work, since they will never need to create them on their own [12].

Nevertheless, there also exist several renowned computer scientists, such as Mitchel Resnick, who strongly assert the importance of a widespread computing education starting from childhood.

According to what Resnick stated during a TED conference entitled *Let's teach kids to code*, which took place in November 2012, writing by means of the language of software, namely coding, is by now not only a skill that is reserved to the buffs or professionals of information technology, but a language that everybody should know, to be able to use technology to one's own advantage. Furthermore, he also stated that nowadays, differently from what happened in the past, programming does not require particular mathematical skills or a complete education in the field of IT.

In order to prove this, he created *Scratch*, a software that allows children to learn the art of programming as if it was a game. Scratch can be used to develop school projects, animated cartoons or videogames, and – Resnick says – "*as kids are creating projects like this, they're learning to code, but even more importantly, they're coding to learn*". Indeed, while learning to code, they are also acquiring many different skills, which may turn out to be useful in different contexts. In particular, writing code is an occasion for kids to make experiments personally, to proceed by trials and errors in search for different solutions, to work autonomously without being helped by an adult, and to develop their own creativity, logical skills, concentration capacity and care for details.

In the United States, the Obama Presidency seems to have committed to Resnick's thesis. In fact, sort of a trend around coding arose in the last few years.

The code.org organization is aimed at teaching children to write code at school. On their website, there is a video entitled *What most schools don't teach*, explaining how education should change, and the organization's endorsers are Bill Gates, Mark Zuckerberg and Jack Dorsey, respectively founders of Microsoft, Facebook, and Twitter.

There exist several websites providing free lectures and tutorials about programming. Among the most popular ones, there are Code School, Treehouse, Code Avengers, Google Code University, just to name a few. Besides that, there also exist a huge number of MOOC (Massive Online Open Courses) on this topic, such as MIT OpenCourseWare, Coursera, Udacity, and Khan Academy.

Moreover, numerous summer camps and summer schools specializing in technology exist in the USA. One of the most well known among them is the InternalDrive's tech camp, founded 17 years ago. They organize IT courses for children with ages ranging from 7 to 18 years old across 27 different states.

In Estonia, the most connected country in the world, where high bandwidth is everywhere and the connection speed is quite high even in the smallest villages, children learn coding at school, as an education in the field of programming is envisaged by ministerial programs, and pupils do their homework almost exclusively by means of a tablet.

Great Britain also accepted the challenge proposed by the USA and Estonia. Here, the year 2014 was baptized as the *Year of Code*, as the teaching of programming started to be proposed right from primary school [7].

In France, on the contrary, IT education has always been limited to illustrating the usage of basic products. Coding is not currently being taught in schools, and the first possibility for students to approach computer science from an active viewpoint is at university, evidently too late for them to be able to make a real impact in this sector.

And what about Italy? Public education is quite behind if compared to other European countries. Nevertheless, numerous universities and private companies are making an effort to improve the situation.

In what follows, we will analyze the specific features of some coding courses taking place in the USA and in some European countries, with a particular focus on the Italian situation.

2.2 Initiatives in the USA

As previously touched upon, many initiatives arose in the last few years throughout the USA, with the aim of promoting the teaching of coding in public schools. We are now going to provide an overview of three of these projects, that we consider particularly significant. More precisely, we are going to illustrate the *Computer Science for All* initiative, and two projects sponsored by leading universities, namely *The Beauty and Joy of Computing* (Berkeley University) and *CS50* (Harvard and Yale universities).

It is important to mention that, in the US, computer science education is also supported by the Advanced Placement (AP) program, an initiative started in 1955 by the College Board, an American not-for-profit organization created in 1899 under the name of College Entrance Examination Board (CEEB), with the aim of expanding access to higher education. This program allows high school students to follow college-level courses and take the respective examinations, providing them with

the opportunity to gain college credit before entering university. Among more than 30 curricula offered by AP, a course dedicated to computer science could not be missing! Indeed, the AP Computer Science Principles course, started in 2016, pursues the objective of introducing students to the central concepts of computing, inviting them to develop an ability to think computationally, considered as a primary condition for success in many scientific disciplines, such as medicine or engineering. This course is based on a precise curriculum framework organized around the introduction of seven key ideas, helping students to gain a solid understanding of computing and facilitating their approach to programming. The principles underlying this framework and the relevant key ideas will be treated with reference to the course offered by Berkeley university, actually taking inspiration from these principles.

2.2.1 Computer Science for All

Computer Science for All is an initiative promoted by President Barack Obama in order to enable all American students in school age to acquire the computational thinking skills that are needed to play an active role in a world dominated by technology. Indeed, more and more business leaders and educators are recognizing the increasing importance of computer science (**CS**) as a new basic skill, that is not only necessary to the professionals of the IT sector, but also can improve the social mobility of people who are active in different sectors.

The need for a more widespread computer science education is quite evident from the current situation of the job market. In effect, in the year 2015, more than 600,000 well-paying tech positions across the USA could not be filled in, and, most importantly, by 2018, 51% of the so-called STEM jobs (where STEM is an acronym for Science, Technology, Engineering and Mathematics) are estimated to be in IT-related fields.

Parents also appear to be aware of this necessity: more than 90% of the parents interviewed state that they would like their child's school to establish computer science courses. Nevertheless, most surveys seem to provide a quite discouraging picture: only 25% of the American primary and secondary schools actually offer programming and coding courses, and in 22 states computer science is still not taken into account in the high school graduation mark.

In addition to the limited availability of this kind of courses, widespread prejudices and stereotypes dissuade many students from attending these courses, which causes remarkable inequalities even among those who would, in theory, be able to access them, making the situation even worse. In particular, statistics show a worrisome lack of interest for computer science from girls and African-American students, which reflects the current composition of the staff inside many

American IT companies, where less than one third of their employees is constituted by women, and less than 3% by African-Americans.

Nevertheless, tech companies play a crucial role in nowadays economy, and CS for All aims at putting all American students in the condition to join and make an impact inside these companies. This is the reason why the government has launched two important initiatives in favor of students, respectively called *ConnectED* and *TechHire*. The ConnectED initiative, started in 2013, is designed to empower an always increasing number of students to have access to latest generation broadband. The President also has established the federal government to allocate more funds to education, in order to improve Internet connectivity and educational technology inside schools. Moreover, he has invited states, districts, businesses, schools and communities to support this cause. The TechHire initiative, instead, started in 2015, is aimed at ensuring that students can easily enter the job market and that companies can find professionals having this kind of skills.

However, CS for All is not only devoted to students, but to educators, communities and policy makers as well. As far as education is concerned, in particular, the number of qualified computer science teachers is not sufficient to cover students' needs, but with such a shortage of instructors in this discipline, there exists the risk not to leave students with satisfactory competences, from companies' viewpoint. Therefore, in favor of teachers and school leaders, the Corporation for National and Community Service (CNCS), in cooperation with the National Science Foundation (NSF), has promoted a new initiative called *Computer Science Teachers AmeriCorps*, with the goal of reducing such an education gap, by providing intensive refresher courses to thousands of computer science teachers over the next few years.

In 2014, more than 60 school districts also replied to the government's urging, and undertook to provide a higher number of students with a chance to learn coding. Both Republican and Democratic state governors have started challenging projects in order for CS skills to become widespread among population. In particular, the City of New York has announced its intent to increase CS learning opportunities for students by means of an ambitious 10-year program. Furthermore, numerous local- and state-level leaders are committing to expand CS competences. For example, the State of Delaware has recently extended IT education to 13 more schools, and has created an online programming course addressed to all students interested.

2.2.2 The Beauty and Joy of Computing

The Beauty and Joy of Computing (**BJC**) is a computer science course provided by the University of California, Berkeley, mainly addressed to undergraduate non-majors, combining a strong focus of the development of programming skills with lectures, readings, and discussions concerning nonprogramming topics, aimed at analyzing the social context where computing has evolved and the drawbacks and limitations that computing implies.

The course is based on seven *Big Ideas*, corresponding to things to learn and six *Computational Thinking Practices*, corresponding to things to do. Figure 2.1 illustrates the prospect in the official website of the course.

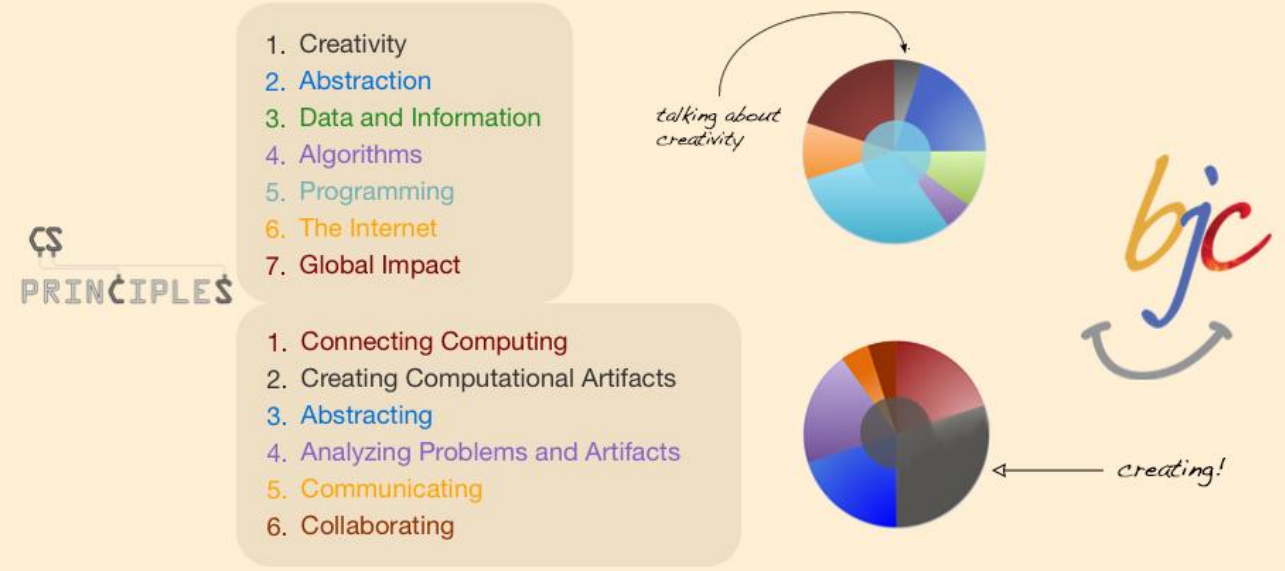

Figure 2.1. Computer Science Principles

The first pie represented in the figure shows the distribution of the topics treated during the course, while the second pie shows how much time students are expected to devote to each practice.

If we take a look at the Big Ideas pie, we can see that BJC is structured as a very ***Programming***-intensive course. The programming language being taught is Snap*!*, because of its capability to combine the ease of use of block languages with more sophisticated features that are typical of text-based languages. Writing code is given such an importance inside this course because it enables problem solving, human expression, and creation of knowledge, by inducing us to translate human intention into computational artifacts. Students come into contact with the main concepts and techniques of programming, developing software, and using software effectively.

Abstraction is a central idea in computer science, and this course aims at emphasizing its importance in the context of programming. It is, indeed, one of the most important problem-solving

techniques, as it helps reducing information and detail to facilitate focus on concepts relevant to understanding and solving problems.

Secondly, this course focuses on ***Global Impact***, in the attempt to analyze the social implications of computing. Without doubt, in fact, computation has modified how people operate, work, live and think. Therefore, the course aims at making students aware of the multiple ways in which computing can be an extraordinary innovation tool. Nevertheless, the course also does not omit to illustrate the harmful consequences that computing may have in some specific contexts, such as, for example, the potential threats to privacy deriving from the use of cellphone cameras, or the tendency to isolate oneself, brought about by people spending most of their time online instead of face to face or, in the case of children, game addiction.

Among the remaining four principles, two (***Creativity*** and ***Algorithms***) are more implicitly than explicitly treated. Of course, students need to exercise their creativity while creating their projects, but the course does not talk about it as much as it does with respect to the other topics. Similarly, while programming, students develop and implement algorithms, but these are not considered as a distinct topic, except analysis of algorithms or asymptotic orders of growth are being treated.

The final set of topics, ***Data and Information*** are considered important and treated exhaustively. Students work with data by using various software applications, in order to gain an understanding of the multiple ways in which an overwhelming amount of raw data can be transformed into information and knowledge. Nevertheless, when possible, this course also faces the topic by proposing programming activities instead of, for example, exploiting commercial software.

Let us now come to the Computational Thinking Practices. With the expression ***Creating Computational Artifacts*** we refer to anything that can be done by means of a computer, such as, for example, creating videos, music, animations, games, websites, programs or spreadsheets. In this course, students display their creativity in the design and development of interesting computational artifacts, and make use of computing techniques to find creative solutions to their problems.

In the same way as the idea of abstraction is a central aspect of the idea of programming, the practice of ***Abstracting*** is also central to programming. In this course, in fact, students make use of abstraction for developing models and representations of natural and artificial events, exploit them for forecasting purposes, and evaluate their effectiveness and validity.

When speaking about ***Connecting Computing***, one could mean connecting it to hobbies, industry, or science, but the aim of this course is to illustrate connections between computing and its social implications. Students analyze these consequences, and meditate on the possible connections between specific aspects of computing.

Almost equally important is the practice of ***Analyzing Problems and Artifacts***, which consists of debugging, understanding what a given code is going to produce, and improving the efficiency of one's programs. In this course, students are taught how to design and implement solutions and artifacts, and, in this context, they need to evaluate the quality of their own projects as well as the computational efficiency of codes written by someone else. Nevertheless, this skill cannot be regarded as an end in itself, but it must be considered as a tool for reaching the final goal of writing correct programs.

The last two practices (***Communicating*** and ***Collaborating***) are also important, but the BJC course does not put great emphasis on them. Communication mainly consists of preparing written and oral presentations of one's own projects, possibly including graphs, pictures, and an analysis of computational efficiency. As far as collaboration is concerned, students in this course are invited to work together in many different activities, ranging from the analysis of specific phenomena based on datasets, to the creation of computational artifacts. With respect to the latter, the BJC course mainly uses pair programming, so as to make students to constantly talk to their partners and share the work [60].

2.2.3 CS50

CS50, started in 2007, is a joint initiative of Harvard and Yale universities, aimed at providing an introduction to the intellectual enterprises of computer science and to the art of programming in favor of comfortable and less comfortable students alike. Since its first appearance, CS50 was available as an open courseware, allowing people to follow the course online for free. Since 2012, it also became available as a MOOC, which means that anyone could not only take the course for free, but also receive a certificate of completion.

From a didactic viewpoint, it is structured as an entry-level course aimed at teaching students how to reason in algorithmic terms and solve problems in an efficient way. The course covers the following main areas: abstraction, algorithmic thinking, data structures, encapsulation in non-Object Oriented Programming (OOP) languages, resource management and efficiency, security, software engineering, and web design. Programming languages taught include C, PHP, JavaScript, SQL, and the main web development tools, such as HTML and CSS. The exercises proposed are mostly inspired to real-world problems relating to cryptography, economics, computer gaming, but also forensics and biology. At the end of the course, students are also required to realize and present a final programming project, which enables them to achieve a wide and secure understanding of coding and computer science in general.

Till nowadays, though, most of CS50's resources have been primarily devoted to students, but the course designers are currently in a way to make them also available, for example, for teachers, so as to provide them with a support in teaching computer science that they can either adopt or adapt for their own students. **CS50 AP**, where AP stands for Advanced Placement, is in fact a sort of free adaptation in favor of high schools satisfying the new AP CS Principles curriculum framework, that we previously discussed with reference to the UC Berkeley course.

The Harvard College Board will soon be launching two courses explicitly addressed to teachers: **AP CS A**, mainly focused on Java, and **AP CS Principles**, a new curriculum framework that will exist alongside AP CS A. Differently from the latter, which has a well-defined syllabus, AP CS Principles simply consists of a framework, which could be implemented by any number of courses, and CS50 AP will be one of those courses, structured in a similar way to the ones provided by Code.org and UC Berkeley. From a curricular and technological viewpoint, CS50 AP will be identically organized to the Harvard course CS50, but will be adapted to high school programs and integrated with additional resources specifically addressed to teachers and opportunities for professional development.

In particular, the program of the AP CS A course is pretty much equivalent to the Harvard course CS50, as it provides students with some basic computer science knowledge, by illustrating fundamental concepts such as problem solving, design strategies, data organization by means of structures, algorithms, analysis of potential solutions with respect to efficiency, and the consequences of computing from an ethical and social viewpoint. Nevertheless, in addition to this, the course also treats object-oriented and imperative programming and design by focusing on the Java language. Overall, AP CS A can be considered compatible with a first-semester, college-level computer science course.

For what concerns AP CS Principles, this curriculum is aimed at teaching the underlying computation principles by adopting a multidisciplinary approach, as it will introduce students to the pragmatic aspects of computing, such as programming, abstract thinking, algorithms, information retrieval from large data sets, but also propose an overview of some central theoretical issues, such as security concerns, the Internet, and the impact of computing on society. This course also allows students to exploit state-of-the-art technologies (such as GitHub [91], a web-based Internet hosting service) in order to realize computational artifacts for both personal and problem solving purposes. With its rigorous but diverse curriculum, this course aims at increasing participation and interest for computer science.

2.2.4 Further initiatives in support of computing in primary and secondary education

In Guzdial and Wilson's paper entitled *How to Make Progress in Computing Education* (2010), the authors report that although state governments spent billions of dollars to finance STEM education, funds explicitly devoted to research on computing education are still inadequate [49]. The National Science Foundation (NSF), aware of this state of affairs, started several initiatives such as CPATH (CISE Pathways to Revitalized Undergraduate Computing Education), BPC (Broadening Participation in Computing), and, most recently, CE21 (Continuing Education for the 21st Century), which have given an important contribution to realizing projects aimed at introducing CS and CT concepts already at secondary level education. Further attempts to boost middle and high school students' interest for CS careers were made by the Defense Advanced Research Projects Agency (DARPA) with the CS-STEM program, and by the Carnegie Mellon University with FIRE (Fostering Innovation through Robotics Exploration).

Notwithstanding ongoing research aimed at informing school curricula with the inclusion of an effective CT teaching, there still remain huge open questions, particularly with respect to the way in which computing teachers should be trained, and the strategies that could be adopted to reduce the gender gap characterizing this industry. With regards to this, in the United States, the Georgia Computes! alliance is committing to prepare teachers, develop CS/CT primary and secondary level curricula, as well as encourage girls to undertake CS careers.

For what concerns curricula, several projects were also started with the aim of introducing CS into schools. One of the most successful initiatives of this kind, at least with regard to the United States, is Exploring CS [70], designed as a 1-year college preparatory course addressed to high school students. Other projects include CS4HS (Computer Science for High School) [73] and Computing in the Core [66], both of which arose from preexisting collaborations between universities, national institutions, and IT companies such as Google and Microsoft. The Computer Science Teachers Associations (CSTA) devised a Model Curriculum for primary and secondary level CS education, providing directions on how to create interest among students and incite them to improve their CS skills. Google also opened an Exploring Computational Thinking [74] section inside its website, containing a large number of links to CT and coding resources present on the web. Finally, the Association for Computing Machinery (ACM) has recently enriched its ACM Inroads magazine with a new thread, called EduBits, that highlights the main educational initiatives being carried out by ACM itself and associated organizations.

2.3 Initiatives in Europe

Even in Europe a large number of parents, teachers, as well as economists and politicians are starting to consider computing and coding skills as a major condition to guarantee the success of Europe's digitalization, economic growth, and the wealth of citizens and society.

In what follows we are going to focus on some important initiatives started in Ireland, France and Italy. In particular, with respect to Ireland, we will provide an overview of the structure and objectives of the Bachelor of Science degree in Computational Thinking that was recently established at the Maynooth University. As far as France is concerned, we will discuss the current situation and present the main commitments undertaken by Hollande's government in the field of computer science education. Finally, we will illustrate some projects started in Italy, namely *Programma il Futuro*, *CoderDojo* and the *Digital Camps* sponsored by H-Farm.

2.3.1 UK and Ireland

In 2012, the United Kingdom promoted the institution of Code Club, an educational program aimed at helping more and more young students to learn how to realize their ideas with code. In 2015 Code Club joined forces with the Raspberry Pi Foundation, a registered UK aid organization. Composed by a nationwide network of educators and volunteers, it consists of a set of free coding clubs addressed to children aged 9 to 11.

Currently, there exist over 8,000 clubs of this kind in over 80 countries, and the relevant projects have been translated in 15 languages.

These projects consist of simple step-by-step guides that allow children to learn to use Python, Scratch, and the markup languages HTML & CSS by creating games, animated stories, and simple websites. These guides are built in such a way as to gradually introduce programming concepts, to allow students to expand their knowledge incrementally. Moreover, this approach also has the advantage that the adult running the session does not need to be a computing expert [61].

In Ireland Maynooth University has recently instituted a BSc degree in Computational Thinking aimed at teaching students how to think, both from a logical and creative viewpoint. Besides computer science courses, the study plan also includes courses in mathematics and philosophy, where the latter is mainly intended to introduce students to the basic concepts of logic and the most relevant theories of knowledge.

From a didactic viewpoint, this degree course allows students to develop the following competences and skills:

- Combining the creativity of human brain with the computational capacities of machines for the purpose of solving problems;
- Exploiting their own experiences to hypothesize solutions in conditions of uncertainty (*heuristics*);
- Simplifying problems by applying models, in order to make them more easily addressable;
- Reformulating apparently complex problems into ones that have already been solved;
- Thinking in abstract terms, in the attempt to identify known patterns inside specific data sets;
- Understanding the theoretical fundamentals of computer science and their mathematical derivation;
- Analyzing the human processes of learning and thinking;
- Developing a robust knowledge of programming and high-level problem-solving skills;
- Acquiring creative and communicative skills.

CoderDojo is another example of how universities and private companies are committing to allow students of primary and middle school to become familiar with the basic concepts of programming. Born in Ireland in 2011, it is now also widespread in many other European countries, such as Italy, Spain, Germany, Belgium and Netherlands. It is an open, autonomous and non-profit movement organized in hundreds of independent clubs located worldwide. Every single dojo must provide the relevant services completely free of charge, in full compliance with the Hello World Foundation Charter for CoderDojo, the international statute of the organization. It can be described as a sort of gym dedicated to boys and girls with ages ranging from 7 to 17 years old, who are interested in learning coding.

2.3.2 France

Europe and especially France suffer a remarkable delay in the field of the IT industry, if compared to the most dynamic countries, such as the United States or some Asian countries. Such a delay can be partially explained by the poor quality of computer science education, which is generally limited to the usage of common software applications, such as Microsoft Office, or web browsers.

Nevertheless, in the last few years, the awareness of the need for an education in computer science from an active viewpoint, and no more only from a passive user's perspective, has notably increased. Moreover, current circumstances appear to be quite advantageous for the introduction of

an authentic education in computer science. Here are some examples of this: the repeated requests of companies for a more advanced education, due to the lack of well-trained employees in the IT sector; the interest that students bring to this discipline, which is an essential part of their everyday life; the possibility to propose examples of application to very diverse and captivating fields; the excellent adaptation to online teaching.

In such a context, the pressure of French IT companies led president François Hollande to announce his ambition that "*France can be leader in the e-education sector*". In March 2015, in fact, computer science and coding were officially registered in the list of high school programs, setting as an objective that "*students must know the principles of programming languages and be able to create computational artifacts by using simple algorithms*".

In reality, already starting in 2013, computer science was included in all higher school preparatory classes, consisting of biennial preparatory courses aimed at training undergraduate students for admission to one of the so-called *grandes écoles* (grand schools), that are higher education institutions outside of the main framework of the French university system. The study curriculum ranges from algorithmics to databases, including one of the most common programming languages. In spite of this, a problem still exists: the French Ministry for National Education, in fact, does not recognize computer science as a school subject; as a result, there are neither teachers, nor continuing education establishments, nor general inspections, nor recruitment competitions, nor teaching qualifications with respect to this discipline.

To help turn the situation around, the institution of a teaching qualification in "*mathematics with option computer science*" has been announced for 2017. Then, there is still a way to go before computer science can be officially included among school disciplines, nonetheless, with the most recent reforms, France has surely made a major breakthrough.

2.3.3 Italy

Even in our country we are witnessing a growing awareness that an understanding of the fundamentals of computer science is essential for a student of nowadays in order to be culturally prepared for any kind of career he will undertake in his future, exactly as it happened in the past for mathematics, physics, biology and chemistry. Moreover, it is now widely known that the cultural-scientific aspect of computer science, which is what we refer to when speaking about computational thinking, helps develop logical skills and improves the ability to solve problems in a creative and effective way, which constitute important qualities of future citizens.

The most effective way of developing computational thinking, when dealing with kids, is to educate them to the art of coding in a playful context. Therefore, starting from the academic year 2014/2015 the Italian Ministry for Education, University and Research has launched an initiative called **Programma il Futuro** (literally "Program the Future"), with the aim of providing schools with a set of simple, funny and easily accessible tools in order to educate students to the basics of computer science. Thanks to this project, taking the cue from the American experience, Italy has been one of the first countries in the world to introduce the basics of computer science and coding as a structural element of education inside primary and middle schools. The tools provided by the Ministry for Education consist of a set of interactive and frontal lessons available online, which can be used as they are or adapted to the specific needs of every educational institution. These resources are designed to also be used by teachers of different subjects, because neither any specific technical skills nor any scientific competences are needed, and the didactic materials can be usefully exploited by students of all ages.

This idea provides for two different kinds of study track: one basic track and five advanced tracks. The basic participation modality, denominated *L'Ora del Codice*, (The Hour of Code), consists of proposing students an hour of introduction to computational thinking. A more advanced participation modality, instead, consists of following up this first introduction with more advanced lessons, always focusing on the theme of computational thinking, whose presentation can be distributed during the whole school year.

In 2012 the CoderDojo initiative was also started in our country, where more than 40 dojos are currently operative, distributed throughout the national territory, and their number is constantly growing. Ca' Foscari University of Venice has also housed several CoderDojos starting from 2015. Thousands of Italian kids have already taken part in one or more meetings organized by the movement, having the chance to learn programming languages such as HTML, JavaScript, CSS, develop simple videogames by using Scratch and practice low-level hardware programming thanks to Arduino.

Moreover, in Rome, Milan and Turin, a start up called Codemotion is active, which provides courses for children inside universities. The company's motto is: *programming is as important as reading and writing*.

Another important initiative realized in Italy is constituted by the Digital Summer Camps organized by H-Farm, a digital platform started in 2005 with the goal of assisting young entrepreneurs with the launch of innovative ideas and supporting the transformation of Italian firms in the sense of a digital perspective. This outdoor summer program offers a wide, varied and interesting proposal to allow kids and young students from 7 to 18 years old to spend their summer

holidays in doing something pleasant and useful at the same time. It consists of ten *all-digital* weeks, in English as well, where students have the chance to go through the possibilities offered by technology to learn in an interactive and never tedious way, while being lodged inside the company's structures.

Each of the ten weeks composing the program is devoted to a specific topic. In particular, the course held during the summer 2016 was organized as follows.

Week 1: Graphic and Web Design;

Week 2: Digital Storytelling;

Week 3: Coding;

Weeks 4 & 5: Digital Fabrication;

Weeks 6 & 7: Minecraft;

Weeks 8 & 9: Robotics;

Week 10: Mobile App Development (in English) or Videogames.

Since 2016 H-Farm has also set up an additional three-weeks course specifically addressed to kids with ages ranging from 3 to 6 years old. The course is aimed at introducing kids to a creative relationship with digital tools, thanks to which they will take the first steps towards coding, electronics and robotics [76].

On the occasion of the semester of Italian presidency of the Council of the European Union and of the Europe Code Week, the Italian Digital World Foundation (Fondazione Mondo Digitale) and the Embassy of the United States to Italy, with the collaboration of the American association Girls Who Code, promoted the first edition of *CodingGirls Rome-USA*, consisting of eight days of events, going from October 12 to 19, 2014, entirely dedicated to female children and girls from primary to high school. This initiative was also supported by the Ministry of Education, University and Research, the city of Rome, the Computer Science Department of the Sapienza University of Rome, and Microsoft.

The event was organized as a sort of relay race among roman schools, which hosted coding laboratories held by the coaches of Girls Who Code and by some female tutors, university students and researchers. The last two days, instead, were dedicated to a *hackathon*, a sort of coding marathon where students were required to create innovative applications able to respond to the challenges posed by the Foundation, some of which also implied *family coding*, involving teachers, mothers and grandmothers of primary school little girls. The working language was English, and the computers used for the challenge were provided by Microsoft [62].

Chapter 3

Block Programming Tools

This chapter will illustrate some of the most popular educational block programming tools used by educational institutions to introduce the students of primary and secondary schools to the basics of computational thinking.

3.1 Introduction to block programming

Block programming languages are a quite recently invented educational tool that allows teachers to provide their students with simplified versions of common programming languages, where each instruction is constituted by a single block, and a set of blocks combined together constitutes a program.

Figure 3.1. Example of a C program written by means of instruction blocks [56]

Block languages were developed with the intent to solve two major problems that arise when teaching computer science to kids, that are:

- The difficulty to remember all the commands of a given programming language precisely;
- The difficulty to apply the exact syntax of the commands of a given programming language.

In particular, educators with experience in this field state that syntax seems to be the greatest obstacle for beginning programmers [64]. Indeed, teaching a new language and a new approach to thinking and problem solving at the same time can result to be quite challenging, because the necessary focus on syntax might distract the attention from the underlying computing concepts. For this reason, several tools have been developed, aimed at avoiding that students need to worry about syntax, while learning the basics of computing. **Scratch** is likely to be the most popular programming environments of this kind, and this is why we will devote to it a large part of this chapter. Nevertheless, we will also provide an overview of some other tools, particularly **Snap!**, a sort of expansion of Scratch, **Stencyl**, mainly focused on games, and the **Blockly** library, created by Google.

3.2 Scratch

Scratch is a free visual programming language developed by the MIT Media Lab [31]. Inspired to the constructionist learning theory, and designed for the education to coding by means of visual primitives, it can be easily used by students, teachers and parents for both pedagogical and entertainment purposes, being suitable for projects ranging from mathematics to natural sciences, and permitting to realize simulations, visual experiments, animations, simple musical compositions, interactive art, and videogames.

Its name was derived from the homonymous technique of *scratching* (i.e., mixing sounds), used by turntablists, suggesting the ease with which multiple sounds can be mixed and projects can be combined together thanks to Scratch. In the slang of computer science, indeed, scratching means to reuse blocks of code that can be effectively exploited for new purposes, easily combined, adapted to different situations, and shared with other users, which constitute some of the key features of Scratch.

Its first version for desktop was released in 2003 by the Lifelong Kindergarten group of the MIT Media Lab, led by Mitchel Resnick, and his consulting firm named Playful Invention Company, located in Montreal and co-financed by Resnick himself together with Brian Silverman and Paula Bonta. Since 2007 an option has been added to allow users to share their projects with other people, and shared projects can be legitimately modified and saved with the relevant changes made by other users. Furthermore, starting from the Scratch 2.0 release, it is also been made

possible to define a given block of instructions as belonging to a specific user, within a complex project developed by several users.

3.2.1 Educational use

Shortly after its first release, Scratch became widely used in the UK thanks to Code Clubs, a nonprofit initiative, founded in 2012, aimed at giving children aged 9 to 11 an opportunity to acquire basic coding skills through free after-school activities. As part of school and college programs, Scratch is generally used as an introductory language, because students can create interesting applications with relative ease, and later apply the learned skills to other introductory programming languages, such as Python or Java.

Even though it is mainly used for creating simple games, Scratch can also prove to be useful for didactical purposes, as it allows teachers to realize conceptual and visual lessons relating to different disciplines, such as mathematics, history, and even natural sciences. With respect to the latter, for example, it can be used to create animations illustrating quite tough concepts, such as Galileo Thermometer, Hooke's Law, or even plant cell mitosis and the water cycle. In the field of social sciences, educators can easily create interactive quizzes, games and tutorials that stimulate children's mind. On the other hand, when used by students themselves, it permits them to create *meaningful personal as well as educational projects* [27], becoming a practical expression tool.

Some leading universities have also opted for Scratch as an introductory programming language. For example, Dr. David J. Malan from Harvard University prefers students to use Scratch for the first week of his basic computer science course, instead of commonly used introductory languages such as Java or C. Nevertheless, Scratch presents obvious limitations when used for college level education. Indeed, Malan himself switches to use C after the very first lectures [68].

3.2.2 Language features and user interface

Scratch provides an object oriented approach to programming, where objects (called s*prites*) can be selected from a list of default sprites provided by Scratch itself, or, alternatively, can be drawn as bitmap images by using a simple editor that is part of the programming environment, or imported from external sources, such as photo galleries or webcams. Nevertheless, whether or not Scratch is an OOP language is a disputable issue. Indeed, it shares some features with common OOP languages, particularly the following:

- the use of objects, represented by sprites;
- the possibility to access the properties of a specific sprite from other sprites;
- limited cloning capabilities, available as of Scratch version 2.0, that enable dynamic sprite generation;
- the possibility to create pseudo-objects by means of lists.

On the other hand, Scratch differs from the standard OOP paradigm by reason of the following features:

- it does not support custom objects;
- before version 2.0, it did not support dynamic sprite generation.

In consideration of this, some scholars agree to consider Scratch an imperative programming language.

The user interface, illustrated in Figure 3.1, is composed, from left to right, by the so-called *Stage* area, located in the upper left portion of the screen, where the results appear and the code comes alive, thanks to the sprites whose thumbnails are listed below the stage. Within this area animations, graphics, etc., everything can be displayed in either small or normal size, or even in full-screen mode. Any point inside the stage is defined by a couple of Cartesian coordinates (x, y), whose origin (0, 0) corresponds to the stage center.

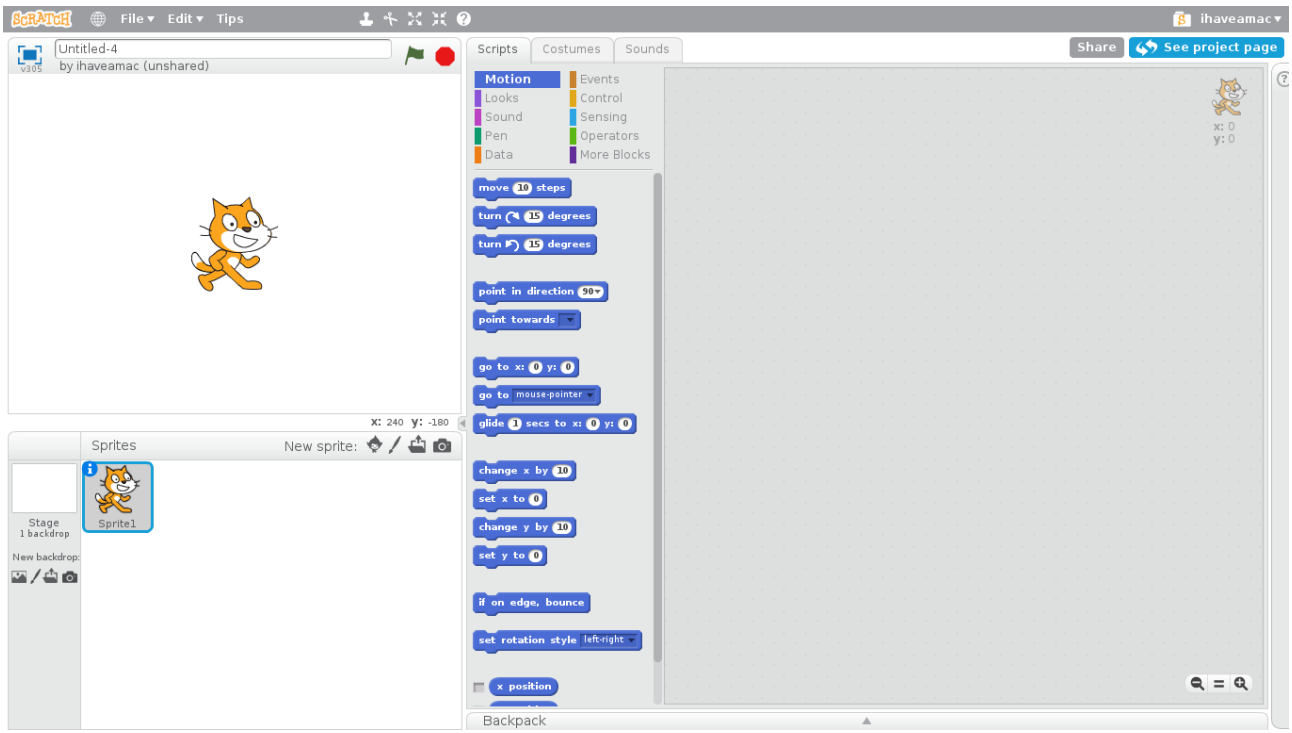

Figure 3.1. Scratch 2.0 development environment at startup

The central portion of the development environment contains the Blocks Palette, where all available commands are listed. The user can apply a chosen block of instructions to a specific sprite by selecting the latter in the lower left area of the screen, and dragging the needed commands onto the grey area on the right. The blocks palette can be retrieved under the *Scripts* tab, providing a subdivision of blocks into the following categories: Motion, Looks, Sound, Pen, Data, Events, Control, Sensing, Operators, and More Blocks. The table in Figure 3.2 illustrates the features and functions of each of these categories.

In addition to the Scripts tab, two further tabs are provided: the *Costumes* tab and the *Sounds* tab, which respectively allow the user to modify the look of a specific sprite, creating animations, and to insert sound effects or music in the application.

Category	Notes		Category	Notes
Motion	Moves sprites and changes angles.		Events	Contains event handlers placed on the top of each group of blocks.
Looks	Controls the visuals of the sprite; attach speech or thought bubble, change of background, enlarge or shrink, transparency, shade.		Control	Conditional if-else statement, "forever", "repeat", and "stop".
Sound	Plays audio files and programmable sequences.		Sensing	Sprites can interact with the surroundings the user has created and can import from PicoBoard or Lego WeDo.
Pen	Draw on the portrait by controlling pen width, color, and shade. Allows for turtle graphics.		Operators	Mathematical operators, random number generator, and-or statement that compares sprite positions.
Data	Variable and List usage and assignment.		More Blocks	Custom procedures (blocks) and external devices control.

Figure 3.2. Block categories in Scratch 2.0 [89]

For what concerns variables, Scratch 1.4 only supported two types of variables, *global* (public) and *local* (private). Starting from Scratch 2.0, a further type was added: *cloud*.

Global variables are those variables whose status has not been changed after creation. In other words, when a variable is created, it is always global. These variables can be read and modified by any sprite, or even the Stage.

Local variables (also called *personal*) are also global as they are created, but a different option is selected in the variable creation dialog: *for this sprite only*, indicating that their content can only

be modified by the relevant owner, while it can be read by all sprites. The Stage cannot own any local variable.

Cloud variables can only store numeric values at present, and are stored on the server. When a cloud variable is modified, the relevant update is valid for all open copies of the project of interest, and is saved in view of the project's next opening. Variables of this type are identified by a small cloud icon next to their names.

Scratch can also be considered an event-driven programming language, as it includes a set of possible events, similar to broadcasts, which can trigger specific scripts. In particular, we list hereafter the four built-in events available in Scratch:

- *When Green Flag Clicked* activates the scripts connected to the relevant block once the green flag has on top of the Stage has been clicked;
- *When () Key Pressed* triggers the scripts related to this event when the specified key is being pressed;
- *When () Clicked* activates the scripts related to this event once the specified sprite or clone of the sprite has been clicked;
- *When I Receive ()* invokes the scripts connected to this event once the specified broadcast has been received by a calling script.

The source code of Scratch versions 0.x and 1.x was implemented in Squeak, which is based on Smalltalk-80. Version 2.0, instead, is developed in ActionScript, with an experimental JavaScript-based interpreter implemented in parallel.

3.2.3 Community of users

The slogan of the Scratch online users' community is *Imagine, Program, Share*, indicating the importance of sharing and of the social implications of creativity in the philosophy underlying Scratch.

Scratch projects are not regarded as "black boxes", but can be easily combined together in order to create new applications. Indeed, any community member can upload a personal project from the programming environment to the Scratch website, and, since projects are open source, users can also download other people's codes for study purposes or for remixing them into new applications.

Members are also allowed to comment, tag, favorite and "like" projects by other developers, follow the activities of other members, and share their own ideas. Chat rooms are not provided yet, but they are likely to be in future releases. Scratch applications can be run in all most common web browsers by installing the Flash Player plug-in.

The website is currently viewed over 150 million times per month and includes about 17 million registered members, 400,000 of which are active users who regularly create and share their projects (so far, almost 20.5 million projects have been published in the Scratch website) [88].

Approximately once a month, the *Scratch Design Studio* is organized, a competition designed to encourage users to create and share basic graphics programs.

In 2008, the Scratch online community platform, called *ScratchR*, was awarded an honorary mention from the Ars Electronica Prix.

There also exists an online community for computer science educators, named *ScratchED*, that allows teachers and school principals to get in touch with each other, share resources and organize group meetups.

3.2.4 Introduction to Scratch programming

In this paragraph and the ones that follow, we will provide a basic introduction to the Scratch programming language by illustrating some of the most common coding techniques used by project developers. In particular, we will now explain how to create a simple *Hello, World!* program.

Once we have started the development environment, we are presented with the default stage shown in Figure 3.1, consisting in the so-called Scratch Cat over a white backdrop.

Figure 3.3. The Scratch Cat

First of all, we might be interested in choosing a different sprite for our simple application. If we explore the *New sprite* menu below the stage, we can see that it provides a set of options suitable for this purpose:

- Choose sprite from library;
- Paint new sprite;
- Upload sprite from file;
- New sprite from camera.

For the sake of simplicity, we will use the sprites available in the Scratch library, by selecting the first option. In so doing, we are presented a set of sprites, from which we might choose any one. For our example, we decide to choose the first sprite listed: Abby.

At this point, we also might want to create a more captivating background. This can also be done by choosing one of the backdrops provided in the Scratch library, by selecting *Choose backdrop from library* from the *New backdrop* menu, also located below the stage. Figure 3.4 illustrates a possible backdrop over which we have inserted the previously chosen sprite.

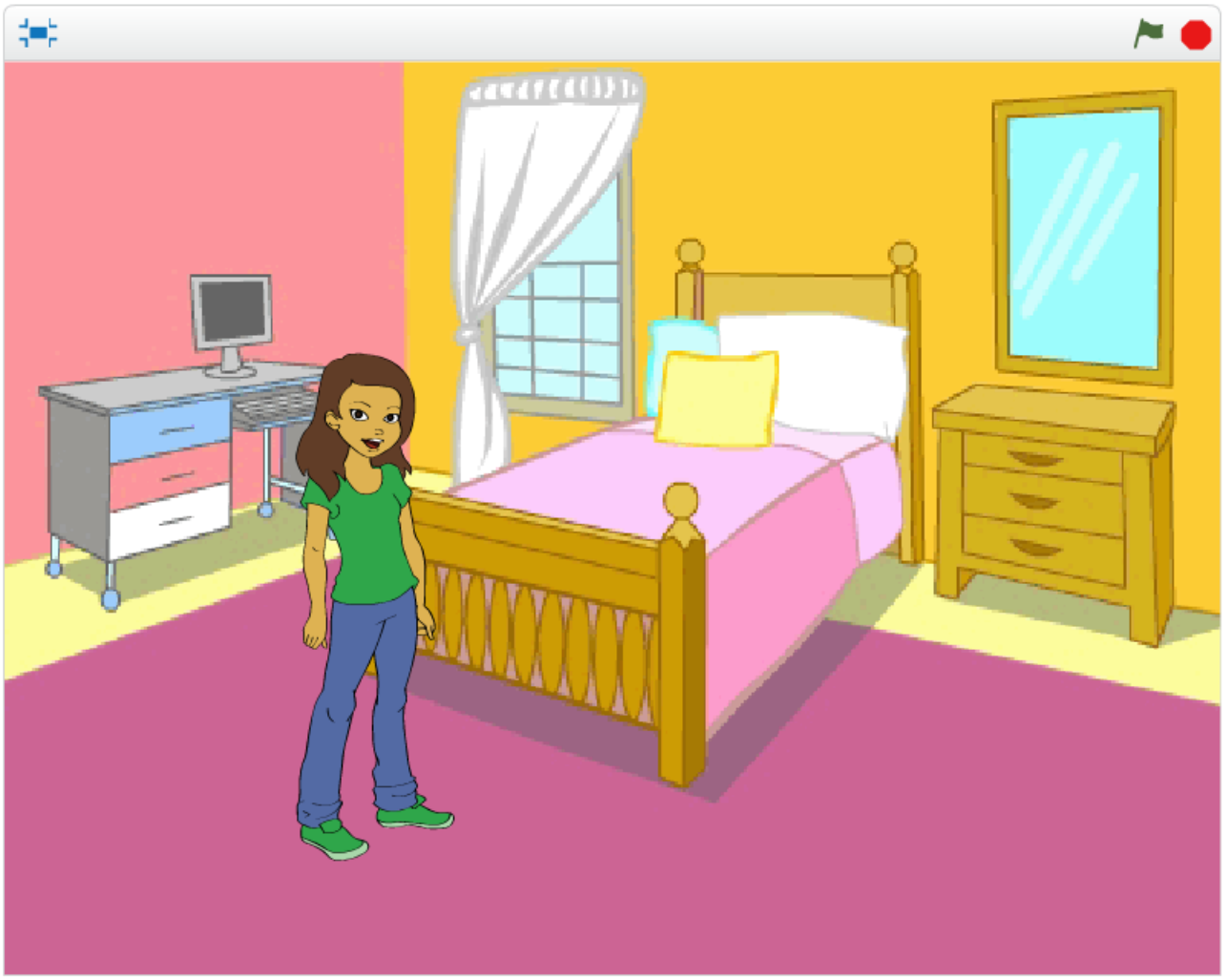

Figure 3.4. A possible stage: Abby in her bedroom

Once we are done with the stage setting, we want to make our sprite say something like “Hello, world! Here’s my bedroom.” To do this, we first need a specified event to trigger our script. Scratch programs are generally started by clicking on the green flag icon at the top right of the stage. Thus, why not to choose exactly this as a triggering event?

To realize this in practice, we need to use the *when green flag clicked* instruction, that can be retrieved among the blocks belonging to the Events category. It is to notice how all the event blocks including the *when* keyword are hat-shaped, which indicates that they always must be located at the top of a script.

To make Abby speak, we need to turn our attention to the Looks category, because the words pronounced by sprites are displayed as speech bubbles. In fact, the instruction named *say "Hello!" for 2 secs* seems to fit our needs, by simply modifying the text and the number of seconds as we wish. Figures 3.5 and 3.6 illustrate the final script and what appears on the stage when clicking on the green flag icon.

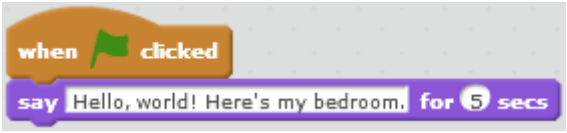

Figure 3.5. *Hello, World!* in Scratch

Figure 3.6. Output of the *Hello, World!* script

3.2.5 Scratch syntax basics

Let us now continue our treatise of the Scratch language by analyzing some aspects of its syntax. For example, we might assume that in Abby's bedroom there is also a boy dancing hip hop. To represent such a scene, we mainly need a further sprite having the features of a boy, ad a hip hop music.

As a boy, we choose the sprite named *Breakdancer1*, which is particularly suited to our purposes as it includes three costumes (shown in Figure 3.7) representing different positions of break dance.

Figure 3.7. Costumes of *Breakdancer1*

As a background music, we select the hip hop music loop available in the sound library.

To convey the impression of dance, we want the boy to repeatedly change costume, which can be done by using the *next costume* block in the Looks category, but to make this happen continuously, we need to include this instruction inside the *forever* block, which can be found under the Control category. In order to make the different costumes more clearly visible, it is better to wait one second (*wait 1 secs* block) before displaying every new costume, and to make all of this to happen as soon as the program is launched, we need to add the already known *when green flag clicked* triggering event, as illustrated in Figure 3.8.

Figure 3.8. Script of *Breakdancer1* (1)

Now, we only need to add music. The script for doing this (Figure 3.9) is quite similar to the previous one, except for what is contained in the *forever* loop. In this case, in fact, the main block is *play sound "hip hop" until done*, available under the Sound category.

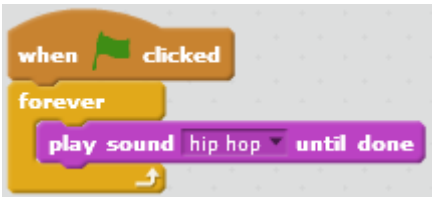

Figure 3.9. Script of *Breakdancer1* (2)

Let us now suppose that in Abby's bedroom there is also a cat chasing a mouse. For simplicity reasons, we will assume the mouse to be symbolized by the computer mouse pointer. By so doing, we will only need to include one further sprite (named *Cat1* in our example) representing the cat. The script associated to the cat (Figure 3.10) also consists in a *forever* loop containing two blocks, taken in this case from the Motion category. In particular, the relevant instructions are: *point towards mouse-pointer*, that makes the cat to point in the direction of the mouse, and *move 10 steps*, that makes the cat not only to point at the mouse while staying in a fixed position, but to also move towards it. Of course, the number of steps, corresponding to the speed of the cat, can be changed to our liking.

A screenshot of our simple application is provided in Figure 3.11.

Figure 3.10. Script of *Cat1*

Figure 3.11. Scratch application screenshot

3.3 Snap!

Snap!, extensively inspired by Scratch, is also a free, block-based educational coding tool designed to allow students to create animated stories, games and interactive applications, while learning the basics of mathematics and computing. Nevertheless, since it expands Scratch's possibilities, it addresses both beginners and more advanced students. Similarly to Scratch, it is completely browser-based and does not require any software installation on the local machine.

Figure 3.12. Snap! language logo

Just like its predecessor BYOB, Snap! 4.0, the stable software release, was developed by Jens Mönig for the Windows, MacOS and Linux operating systems, and has been used for the

introductory course in computer science named *The Beauty and Joy of Computing* provided by the University of California, Berkeley. The relevant design choices and the available documentation are due to Brian Harvey, lecturer at the same university. As of 2015, the city of New York has introduced the Berkeley course in the context of the AP Computer Science Principles curriculum framework in 100 high schools.

Differently from the latest release, the earlier 3.x versions are open-source. By using a license made available by the UC Berkeley website, users can freely download the software source code from the Snap! website itself or from Github, and are allowed to modify it for non-commercial purposes.

Implemented in JavaScript using an HTML5 Canvas API, Snap! 4.0 is compatible with iOS, Windows, MacOS and Linux platforms.

The user interface is organized in the way it used to be in the Scratch releases prior to the 2.0. As shown in Figure 3.12, the leftmost part of the screen includes the blocks palette, the central part contains the scripts associated to a selected sprite, and the rightmost part includes the stage area, featuring the results, and the available sprites' thumbnails.

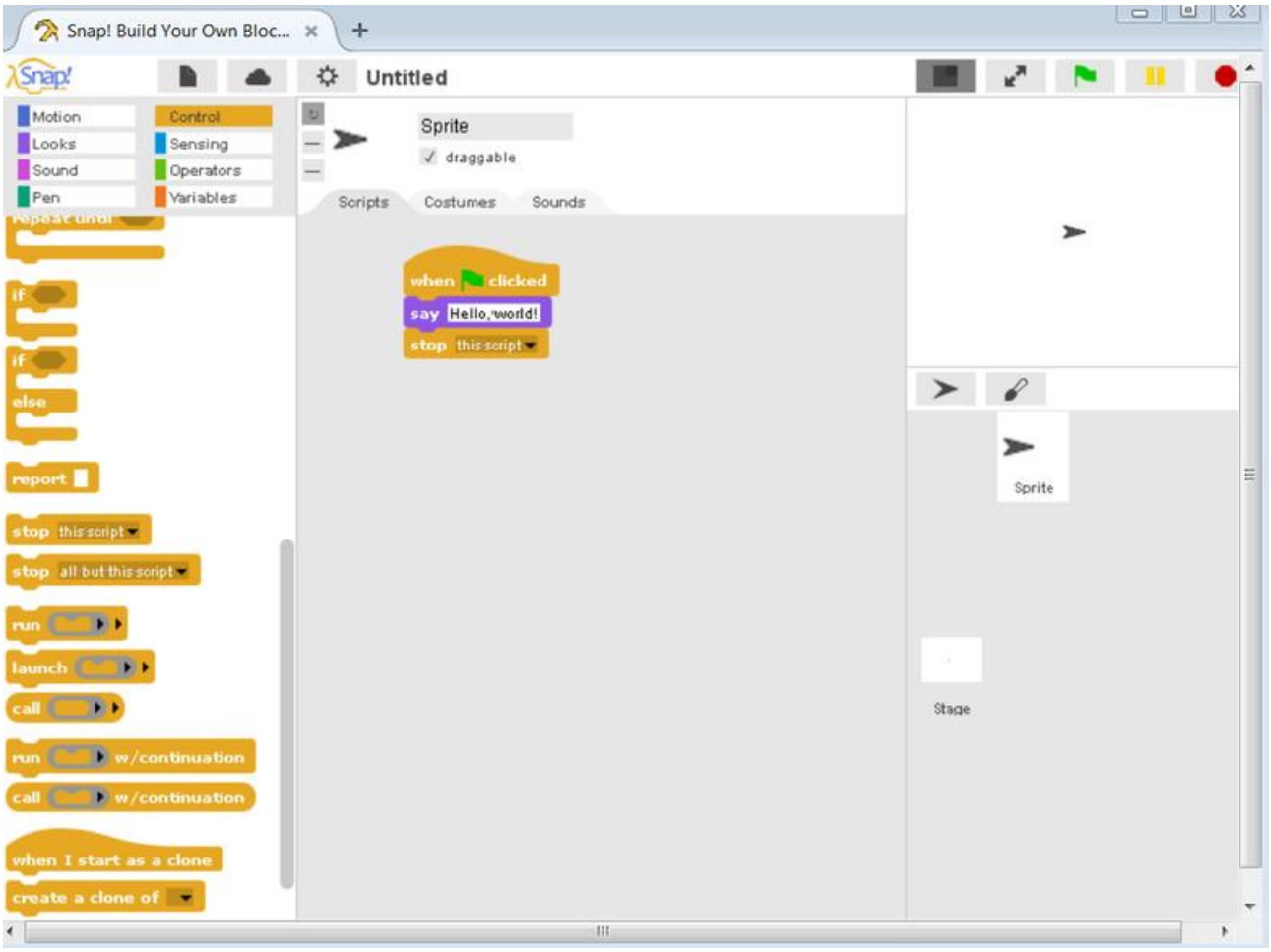

Figure 3.12. Snap! 4.0 development environment at startup

Similarly to what happens in Scratch, Snap! blocks are also subdivided into a set of categories: Motion, Looks, Sound, Pen, Control, Sensing, Operators, and Variables. Moreover, in addition to the Scripts tab, two other tabs are provided: the Costumes tab and the Sounds tab. Nonetheless, compared to Scratch, Snap! provides two less block categories: Events, which is included in the Control section, and More Blocks, which is missing.

A fundamental difference between Snap! and Scratch is represented by the respective paradigms. In fact, while Scratch is essentially an imperative language including some basic properties of object oriented languages, such as the possibility to create and clone sprites, Snap! can be considered an OOP language in many respects, as it does not only conserve the use of sprites, already present in Scratch, but also adds some further features allowing OOP, such as the use of lambda expressions (also called anonymous methods), which constitute a compact way of passing around behaviors as if they were data. Coherently with the OOP paradigm, in fact, a Snap! class can be defined by creating a procedure which reports another procedure. The reported procedure will behave as an instance. It will be able to receive some data, assign specific values to its arguments, and return the appropriate values based on the received data. The script in Figure 3.13 defines a Counter class and specifies the relevant methods to increment, decrement, and retrieve the counter value. This implementation of OOP is made possible by the use of *closures*, which permit procedures to store variables that are located outside the lambda expression, even though these variables will keep existing, with the possibility to be referenced and assigned to. Consistently with its being object oriented, Snap! is able to recognize variable types. More specifically, it distinguishes the following: number, text, Boolean, list, command, reporter, predicate, and object.

Another fundamental design principle is that all data types are *first class*, which means that data of any type can be:

- The value of a variable;
- An input to a procedure;
- The value reported by a procedure;
- A member of an aggregate;
- Anonymous (not named).

In other words all entities support the operations that are generally only available to different entity types. This principle implies that Snap! users do not need to program many special-case procedures themselves, and are able to freely exploit this feature of the language. In summary, Snap! differs from Scratch because of the following main characteristics:

- A higher adherence to the OOP paradigm;
- The existence of first-class functions, lists, and sprites;

- The existence of nestable sprites;
- The possibility to encode Snap! scripts to mainstream programming languages such as C, JavaScript, Python, etc.

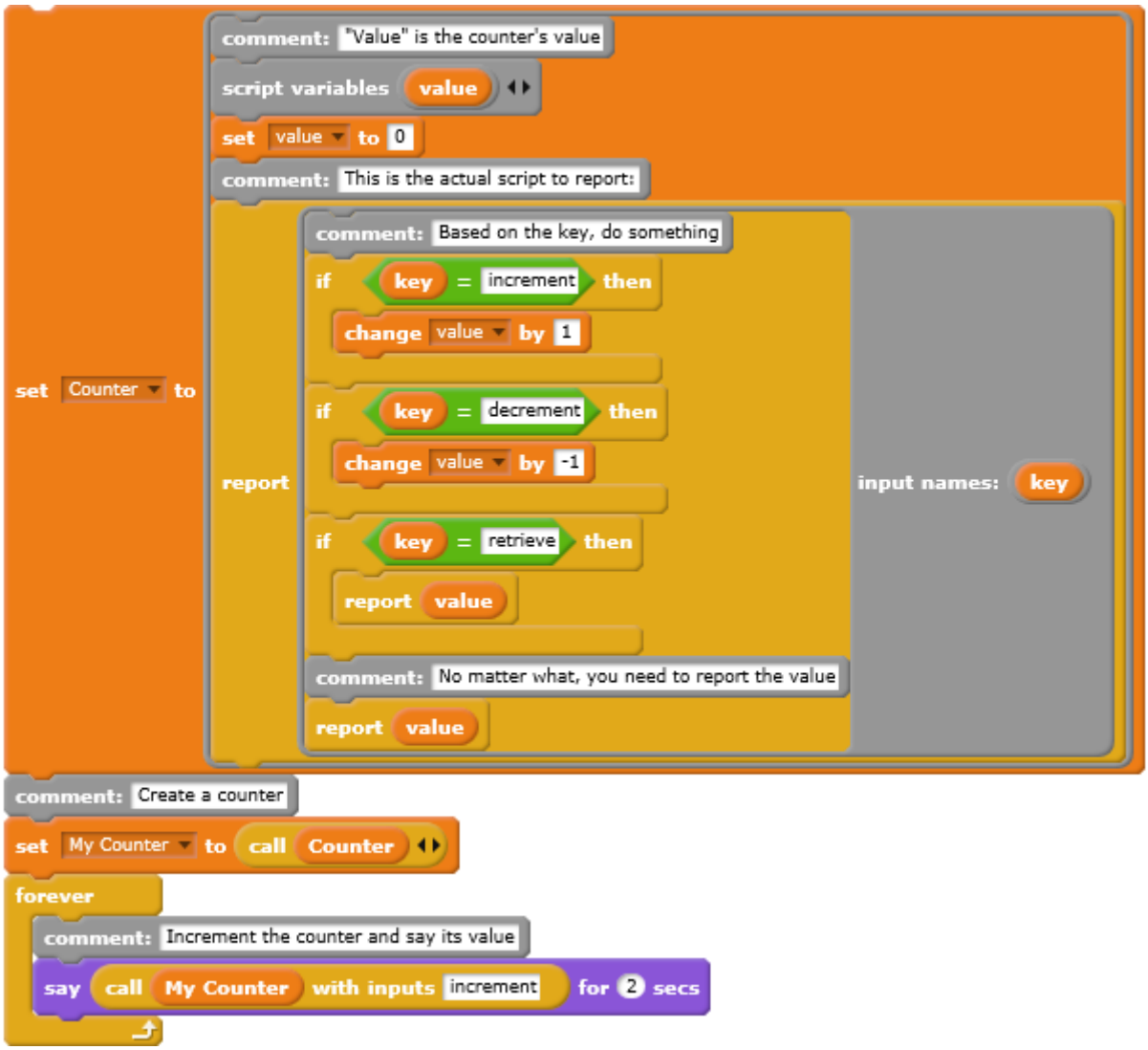

Figure 3.13. Script defining a Counter class in Snap!

3.4 Stencyl

Stencyl is a free game creation software developed in 2011 by Jonathan Chung, and designed to allow users to realize videogames that can be run on computers, mobile phones, and any device connected to the Internet. The original version was called *StencylWorks*, but its name was already shortened to just *Stencyl* starting from the second release.

Games created by using Stencyl can be published on the web thanks to Adobe Flash Player, and can be exported to personal computers as executable files, as well as to several mobile devices as iOS or Android applications. In those games that require full physics simulation, gravity and

collisions are managed in a credible way by Box2D, a free and open source 2-dimensional physics simulator engine developed in C++. Nonetheless, users are allowed to partially or entirely disable the Box2D plug-in, in order to avoid any possible negative impact on the application's performance when dealing with games that do not require physics to be treated realistically. Furthermore, to ensure game creators a flexible WORE (Write Once, Run Everywhere) development style, starting from version 3.0, Stencyl source code and user projects are implemented in Haxe, a cross-platform programming language able to produce source code that is compatible with many different platforms from a single base code, and use OpenFL, an open source game framework permitting to create multi-platform applications. Indeed, Stencyl currently supports several different operating systems: iOS (iPhone/iPad), Android, Windows, Mac, and Linux.

Stencyl is an authoring system, that is a program which allows a user with limited coding skills (usually an instructor or technologist) to easily develop software thanks to the programming options provided by the system. It can be considered an IDE (Integrated Development Environment) as well, that is a software providing computer programmers with comprehensive facilities for software development, since it includes different modules that allow the user to take care of the various aspects related to game creation. In particular, it includes the following editing tools.

- The Behavior Editor allows to write code and manage game logic thanks to modular blocks of instructions referred to as behaviors and events.
- The Tileset Editor allows to import and create tilesets, by managing collision shapes and specifying their appearance and animations.
- The Actor Editor allows to create game entities (Actors) and define their settings with respect to physics, animations and behaviors.
- The Scene Designer allows to create different game levels (Scenes), each characterized by a specific set of actors, tilesets, and behaviors.

Furthermore, Stencyl also includes additional tools that permit to import images to be used as foreground or background objects, import and change fonts, import audio files, and modify game settings, for example with respect to player commands and screen resolution. In order to reduce the need to create common behaviors from scratch, Stencyl provides a library of common game behaviors, and a set of kits containing useful starting points for the most common game types.

When a new behavior is to be created, the user is presented the option to either access Code Mode or Design Mode. Code Mode allows users to program game logic by writing textual code, in accordance with a traditional approach. Alternatively, using Design Mode offers the advantages of a typical VPL (Visual Programming Language). Design Mode users, in fact, do not need to worry about the syntax of a specific programming language, since code blocks can be dragged and

dropped from a palette, similarly to what happens in Scratch and Snap!. These instruction blocks can snap together and nest with one another, giving rise to complex behaviors created from basic components. Of course, syntax errors can be avoided thanks to the fact that not all blocks can snap together. For example, if a space requires a numeric value to be inserted, it will not accept a block representing a string. Because of this, we can assert that Stencyl is a strongly typed language. To make things easier, different block types correspond to different shapes. The table in Figure 3.14 lists the data types supported in Stencyl Code Mode with the corresponding types in Haxe.

Attribute Type	**Haxe Type**
ACTOR	Actor
ACTORGROUP	Group
ACTORTYPE	ActorType
ANIMATION	String
BOOLEAN	Bool
COLOR	Int
CONTROL	String
FILTER	Dynamic
FONT	Font
IMAGE	BitmapData
IMAGE_INSTANCE	BitmapWrapper
INT	Int
JOINT	B2Joint
LIST	Array
MAP	Map<String, Dynamic>
NUMBER	Float
OBJECT	Dynamic
REGION	Region
SCENE	Scene
SHADER	Dynamic
SOUND	Sound
TEXT	String

Figure 3.14. Stencyl types with the corresponding Haxe types

Stencyl can also be considered an event driven language, as it allows to create custom events, namely events that only happen when the user tells them to happen. We might be interested in doing this because of two main properties:

- Reuse logic: instead of copying and pasting a given portion of code, we can simply trigger the corresponding custom event;
- The possibility for behaviors to cause stuff to happen.

With reference to programming, custom events are equivalent to the notions of *messaging* or *indirect invocation*; we cannot treat them in the same way as function calls, as we are not always specifying a receiver.

Events represent the building blocks of behaviors, that are reusable, configurable *abilities* which can be associated to Actor types or Scenes, and together constitute the *brain* of a game, managing all interactions occurring within the game itself. Like events, behaviors can be reused and customized individually for different Actor types or Scenes. For example, if we wanted to associate a walking behavior to a given actor type, we should configure its walking speed. The reusability of events and behaviors within Stencyl can be compared to some methodologies provided by traditional OOP languages.

Starting from the 3.0 release, Stencyl was translated in other languages thanks to a crowd-sourced translation effort. Users are allowed to view single phrases and words, submit their own translations and assign a vote to other users' translations. As a result, starting from June 2013, the software has been fully translated into five languages (Chinese, German, Italian, Slovak and Spanish) and is currently partially available in 21 other languages as well.

3.5 Blockly

The Blockly library, implemented in JavaScript and created by Google as a tool for computing education, is aimed at making programming easier by providing a visual code editor inside web and Android development environments. In this editor, concepts like variables, logical operators, conditional expressions, loops, etc. are represented as graphical blocks which may or may not interlock together, depending whether a given set of blocks is syntactically correct or not. In such a way, it allows users to code by using the language they need to, without worrying about respecting syntax.

Blockly is targeted to developers. Nonetheless, Google also created a set of educational apps, called *Blockly Games*, that teach programming basics to children who have had no prior experience

with coding, to put them in a condition, after these games, of using conventional text-based languages.

From a developer's perspective, instead, Blockly mainly constitutes a text box containing syntactically correct portions of code, which can be exported to several programming languages, including the following widely used options:

- Dart
- JavaScript
- Lua
- PHP
- Python.

As such, Blockly is not a programming language, and its behavior largely depends on the structure of the cross-compiled language. The most commonly used output language is JavaScript, but it we wanted to export our project to a different language, we should not try to preserve the same behavior across both languages. For example, JavaScript assigns empty strings value 0 (false), whereas Lua assigns them value 1 (true). Due to this language dependence, any attempt to define a single pattern of behavior for a Blockly project, with the idea that it will work correctly regardless of the target language, would make our code unmaintainable in most cases.

For what concerns type checking, Blockly can be considered a dynamic-typed language, such as JavaScript and Python, as it lacks the rigid discipline that is typical of static-typed languages, such as Java and C++. More simply, any variable can be assigned any value, since variable definitions do not include any type specification, such as *Boolean*, *integer*, or *float*. In any case, some nonsensical block combinations are prevented from being constructed thanks to the possibility to label connections with type information, so that invalid connections will not fit together. In this way, the user receives an immediate feedback, avoiding many possible oversights. In Figure 3.15 a JavaScript code is presented, that sets the output type of a given block to the string *Number*. 


```
Blockly.Blocks['math_number_property'] = {
 init: function() {
   // ...

this.appendValueInput('NUMBER_TO_CHECK').setCheck('Number');
   this.setOutput(true, 'Boolean');
 }
};
```

Figure 3.15. JavaScript code specifying the output type for a block (Blockly)

Blockly also allows to opt for an event driven approach to programming thanks to the use of *event handlers*, namely functions that get called by the program or the operating system. Such blocks can either wrap the stack of instructions to be executed, or may be headers sitting on top of a stack of instructions, as illustrated in Figure 3.16. In some cases, it is also possible to add a sort of hat to the top of event blocks, as illustrated in Figure 3.17, so as to make them look distinct from other types of blocks.

Figure 3.16. Default event blocks (Blockly)

Figure 3.17. Hat-shaped event blocks (Blockly)

In order to create a Blockly application, a user will have to follow this procedure:

1. **Import the Blockly editor**, which consists of a toolbox storing all available block categories, and a workspace where these blocks can be arranged.
2. **Create the blocks building the user's app**, that will be integrated into the Blockly toolbox.
3. **Develop the app**, by using the code blocks generated through Blockly.

Compared to other visual programming environments that provide similar services, Blockly users can benefit from the following main advantages:

- **Code exportability**: users can easily export their block-based applications to common text-based programming languages.
- **Open source**: the whole code building Blockly can be hacked and used inside any user's web or Android apps.
- **Extendibility**: Blockly can be adapted to fit users' needs, for example by adding or removing specific custom blocks.
- **High capability**: being developer-targeted, Blockly allows the implementation of complex tasks, coherently with common text-based languages.
- **Internationality**: Blockly is currently available in over 40 languages, including right-to-left ones like Arabic and Hebrew.

Figure 3.18 illustrates the Blockly development environment of a simple *Hello World!* application exported to JavaScript.

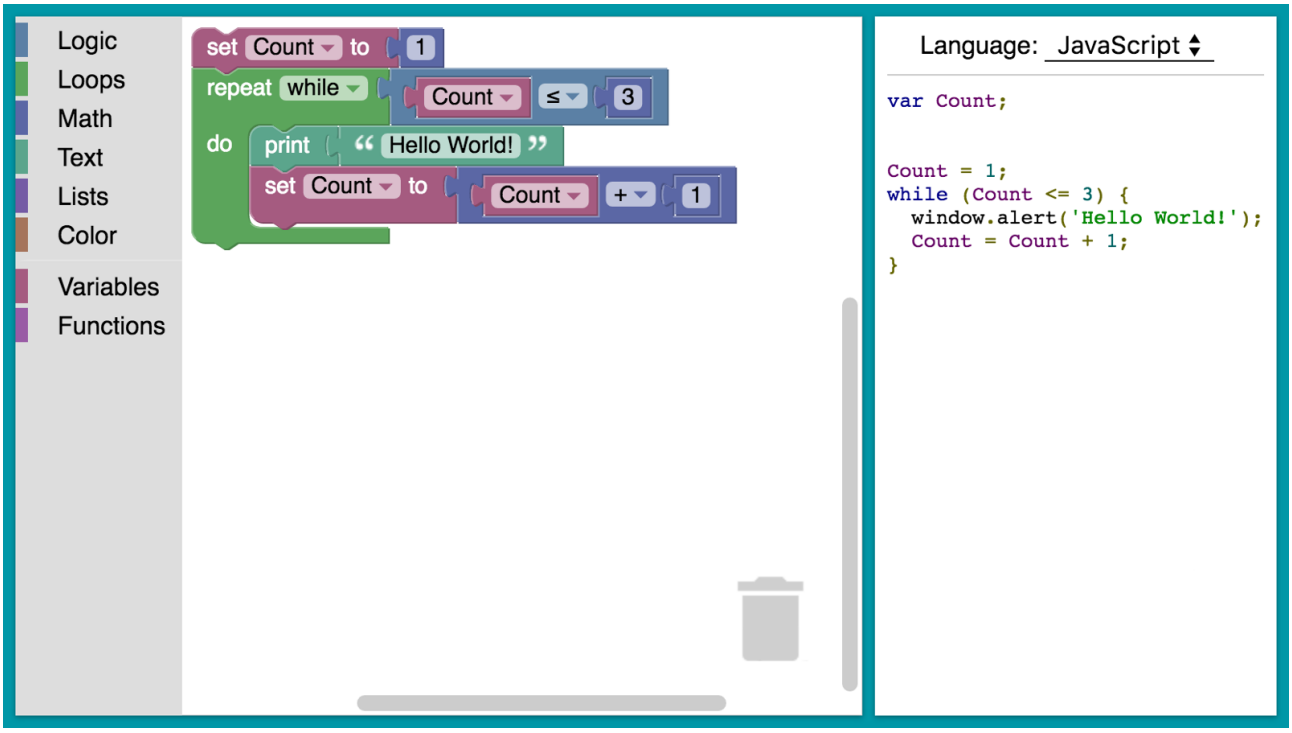

Figure 3.18. The Blockly development environment

3.6 Comparative table

As a conclusion of this overview of the most commonly used educational tools, we present hereafter a comparative table of the described tools, allowing for a rapid examination of the main features of every analyzed software. In particular, for each of them, we decided to specify the following features:

- the age range for which a given tool is mainly designed;
- the developer;
- the operating systems it supports;
- the possibility or not to encode scripts to mainstream programming languages;
- the implementation language;
- the typing discipline;
- the possibility to support event driven programming;
- the relevant software license.

	Scratch	Snap!	Stencyl	Blockly
Student age range	8 – 16	12 – 20	14 – 20	9 – 11
Developer	MIT Media Lab	Brian Harvey and Jens Mönig	Jonathan Chung	Google
Operating Systems	Windows, MacOS, Linux	Windows, MacOS, Linux, iOS	Windows, MacOS, Linux	Windows, MacOS, Linux, iOS, Android
Codification to mainstream languages	No	Yes	No	Yes
Implementation language	Squeak (Scratch 0.x, 1.x) ActionScript (Scratch 2.0)	JavaScript	Before v3.0: Java, ActionScript 3, Objective-C, and C++. Version 3.0 and later: Haxe	JavaScript
Paradigm	Imperative with some OOP features	Object oriented	Multi-paradigm (only valid for Code Mode)	Depending on the cross-compiled language
Typing discipline	Dynamic	Dynamic	Static, dynamic via annotations, nominal (only valid for Code Mode)	Dynamic
Event driven programming support	Yes	Yes	Yes	Yes
License	GNU General Public License and Scratch Source Code License	Affero General Public License	Proprietary commercial software	Apache 2.0 license

Figure 3.19. Comparative table of the analyzed programming tools

From the table we can see that, for what concerns the age of students, Scratch and Blockly are mostly addressed to young learners attending primary and middle school, while Snap! and Stencyl are also suitable for being used in secondary and upper secondary level education. Snap! and Blockly, differently from the other tools analyzed, possess the useful feature of allowing for

codification to mainstream programming languages. The used paradigms vary a lot across these tools. Moreover, both Stencyl and Blockly support more than one programming paradigm. In the case of Stencyl this can be done thanks to its multi-paradigm approach, whereas in Blockly the paradigm is related to the cross-compiled language. The typing discipline is dynamic in all cases except for Stencyl, which adopts a static type system. Finally, all described tools provide some kind of support for event driven programming.

Chapter 4

Research on Computing Education: An Overview of the State of the Field

In an influential paper over computational thinking dating back to 2006, Jeannette Wing asserted the importance of this new competency as a primary component of STEM (Science, Technology, Engineering, Mathematics) learning for all children in their school age. The purpose of this chapter is to provide an overview of the current debate concerning the role of CT inside primary and secondary education, by analyzing some recent proposals that rose as a reaction to Wing's article, and to suggest some possible directions for future enquiries.

4.1 Introduction

In March 2006, Jeannette Wing wrote in her article *Computational Thinking* that "*it represents a universally applicable attitude and skill set everyone, not just computer scientists, would be eager to learn and use*" [53].

Following Wing's article, a broad community of educators, researchers and policy makers started to bring more attention to the concept of CT and research on this topic. Furthermore, in 2010, a harsh report by the Association for Computing Machinery provided evidence that more than two thirds of American high schools relegated CS education at quite basic levels, and that the percentage of women employed in the IT field is extremely low [50]. Concerns deriving from these

statistics are well justified if we consider the predictions given by the Bureau of Labor Statistics, stating that computing will constitute one of the fastest-growing job markets until at least 2018 [92]. Nevertheless, since the beginning of the 21st century, the academic community started to see computing education as a core component of STEM learning, because of the countless possible applications of CT and CS to STEM disciplines [20].

Indeed, the idea of a CT education addressed to non-computer scientists is not that recent. Already in the 1960s, Alan J. Perlis discussed the importance for college students of all study courses to acquire basic programming skills and to learn the so called *theory of computation* [17]. In the 1980s, Seymour Papert was the first supporter of the idea of teaching CT to children, which he carried out by creating the LOGO educational programming language [38, 39].

In what follows, we provide a review of the recent research investigating CT, including those schools of thought that are not really in support of teaching procedural thinking to children, but are anyway favorable to developing their digital competencies in wider terms.

4.2 The debate over CT – What and Why?

Wing's recommendations to provide a CT education in schools represented the starting point for two important workshops organized by the National Academy of Sciences, during which leading researchers in the fields of education sciences and computer science met together with influential personalities from the computing industry in order to discuss about the nature of CT, its implications on cognitive processes [34], and its pedagogical aspects [35].

The first workshop focused on reviewing some early notions of CT that appeared to be too tightly bound to programming. Though maintaining their validity, in fact, these notions were broadened to include several computing concepts that actually refer to much more than *just programming* [38, 39]. The workshop had the merit of increasing the consent of the academic community towards CS teaching, although some central questions remained unanswered. For example, there was not a general agreement with respect to how an effective pedagogy should be for educating children to think computationally, or whether an actual separation of CT from programming and computers may exist or not. The aim of the second workshop was, in fact, to reconsider some of these questions by collecting and synthesizing opinions from educators who already covered CT at a primary and secondary school level, in order to share some examples of best pedagogical practices for teaching CT to children.

The definition of CT itself was the object of several recent debates. In an article published in 2011, Wing reformulated her own definition as such: "*Computational thinking is the thought*

processes involved in formulating problems and their solutions so that the solutions are represented in a form than can be effectively carried out by an information-processing agent [55]".

In 2012, some attempts were done to further simplify this definition. For example, Alfred V. Aho from Columbia University defined CT as the set of thought processes involved in problem formulation such that "*their solutions can be represented as computational steps and algorithms* [1]".

Moreover, the Royal Society also proposed a succinct but nonetheless effective definition: "*Computational thinking is the process of recognizing aspects of computation in the world that surrounds us, and applying tools and techniques from Computer Science to understand and reason about both natural and artificial systems and processes* [44]".

Another valuable specification of the essence of CT is the one coming from the CS Principles course that, as we said in Chapter 2, is based on seven *big ideas* related to computing and a set of corresponding *practices* pertaining to computational thinking.

Furthermore, Barr and Stephenson [2] provided a similar *operational definition of CT*, including both a complete enumeration of the possible meanings of CT and a checklist of the core concepts and skills that students are expected to acquire, along with examples and recommendations about the way they might be contextualized and applied to multiple academic fields.

In 2000 Andrea diSessa [10] from the University of California, Berkeley, preceded Wing's call to action by introducing the notion of *computational literacy*, which comprises both the usage of material tools such as coding environments and the cognitive and social implications of computational literacy. Although the term *computational thinking* is currently more habitual than *computational literacy*, probably because of its susceptibility to be confused with earlier ones such as *computer literacy*, *digital literacy*, and *information literacy*, in academic research nowadays the two sayings are often interchangeable.

In 1980 Beau A. Sheil [45] from the Xerox Palo Alto Research Center (PARC) proposed the notion of *procedural literacy*, a further declination of CT that can be described as the application of CT to the creation of video games and other computational artifacts involving new media art and design.

For what concerns the elements which compose CT and constitute the foundations of all curricula aiming to enhance and assess its learning, there is a general agreement among researchers and CS teachers over the paramount role of *abstraction*. Described by Wing as the capability of *defining patterns, generalizing from specific instances* [55], it is valued as CT's keystone, in consideration of its importance in dealing with complex problems. Besides abstraction, today the following aspects are also unanimously accepted to be comprised in the concept of CT:

- Systematic information processing;
- Symbolic representation of complex systems;
- Notions of algorithm and control flow;
- System decomposition or modularization;
- Problem solving by iterative, parallel and recursive way;
- Conditional logic;
- Fulfillment of efficiency and performance requirements;
- Systematic debugging and error detection.

In recent years numerous initiatives have risen with the aim of introducing the basics of CT without the use of a computer. CS Unplugged [65] is one of the best known projects of this kind. Nevertheless, these experiments have made the importance of coding even clearer to educators, not only as a fundamental CS skill supporting the cognitive processes required by CT, but also as a tool for the assessment of computational competencies. Indeed, avoiding the use of a computer does not allow students to face the issues taking place in CT's common practice.

In conclusion, albeit the academic community widely agrees with respect to the pervasiveness of computing in many disciplines, the importance it should be given inside the school curriculum is still far from being clear. Furthermore, the multiple interpretations of CT and the lacking clarity with regard to what should be included in a CS coursework triggered numerous criticisms from educational institutions, and the convenience or not to make all school-aged children develop computational skills, including those who do not show any interest in pursuing a career in the IT sector, remains under discussion. Besides, even if we assume that CT must actually be incorporated in the school curriculum, there is still lack of a general agreement whether it should be given the status of an additional discipline, or dealt with as a single topic in the context of a specific subject (in this case, it is unclear which subject CT should be associated to), or even considered as a multi-disciplinary topic [35]. Finally, researchers are also wondering whether CT is basically distinct from other forms of thought that students can develop thanks to common disciplines such as mathematics and sciences. With regards to this, supporters of CT education respond that, even though it shares some features of mathematical and scientific thinking in general, it also constitutes a unique way of applying each of these thinking skills [46]. Denning and Freeman, for example, point out that although computing *contains echoes of engineering, science, and mathematics, it is distinctively different because of its central focus on information processes* [9]. Similarly, Grover and Pea [15], from the Stanford University School of Education, claim that CT's specific approach to problem solving does not find any actual replacement in the set of skills that children are allowed to develop through mathematics and sciences, and believe that basic CS literacy would put the

adults of tomorrow in a better position to make the most of a world where computing is ubiquitous. Moreover, starting to face CT from early childhood seems not only to reduce the difficulties that undergraduates are known to experience at the beginning of university CS courses, but also increase their interest for this professional sector.

The table in Figure 4.1 summarizes the different approaches we have illustrated.

Workshops	National Academy of Sciences, USA (2010)	Review of early notions of CT that focused on *just programming*.
	National Academy of Sciences, USA (2011)	Sharing of best practices of pedagogies and environments for teaching CT.
Definition of CT	Jeannette M. Wing (2011)	Thought process involved in formulating solutions that can be carried out by an information-processing agent.
	Alfred V. Aho (2012)	Thought process involved in formulating solutions that can be represented as computational steps and algorithms.
	Royal Society, UK (2012)	Process of recognizing computation in the world that surrounds us.
	CS Principles course, USA (2016)	CT as a set of *practices* that correspond to the seven *big ideas* of computing.
	Barr & Stephenson (2011)	*Operational definition*, including core CT concepts and capabilities.
Alternative approaches	Andrea diSessa (2000)	*Computational literacy*, comprising both the "material" tools and the "cognitive" aspects of computing.
	Beau A. Sheil (1980)	*Procedural literacy*, namely the practice of CT in the context of new media art and design.
Pedagogical choices	Can CT be introduced without the use of a computer?	
	Is it convenient to make all children develop computational skills?	
	Should CT be a discipline on its own?	
	Which discipline should CT be associated to?	
	Is CT basically distinct from other forms of thought?	

Figure 4.1. Recapitulatory table of the debate over CT

On the whole, however, we are witnessing an increasing trend to make programming a more widespread skill for everybody. In addition to the numerous initiatives that we illustrated in the second chapter, we could even mention that in Israel an exemplary compulsory high school CS curriculum has been launched starting from 2012, and countries like Australia, New Zealand, Russia and South Africa have included CS in their primary and secondary school curriculum.

4.3 Relevant research on CT education

While until the 1980s the main purpose of research in this field was to find a useful operational definition of CT, in the last three decades literature has shifted its focus to more pragmatic issues, such as how to encourage and evaluate its learning, as well as how to teach CS and programming in particular. Nevertheless, the greater part of research on CS education concentrates on undergraduate classrooms. In what follows, we will analyze the most recent approaches and achievements with respect to CS education addressed to students of primary and secondary school.

4.3.1 Coding tools and research on interaction design

Even at the times of the very first coding environments, such as LOGO, the creation of this kind of tools was inspired to the guiding principle expressed by the phrase *low floor, high ceiling*, which essentially means that beginners should be able to easily take the first steps and create working applications (low floor), but, at the same time, these tools should be extensive and powerful enough to meet the requirements of experienced programmers (high ceiling). An effective coding environment should have both of these features [41]. Indeed, the numerous programming tools that have been created in the last three decades aim at fitting these criteria, even if they do not satisfy them all to the same degree.

Graphical coding environments such as Scratch, but also less known tools such as Alice[4] [57], Game Maker[5] [71], Greenfoot[6] [75], and Kodu[7] [82], mainly focus on the ease of use, as they allow beginner programmers to concentrate on the application design rather than on syntax issues. Other introductory authoring tools adopt the so called *use-modify-create* paradigm taking inspiration from

[4] Alice is a freeware object-based educational programming language with an Integrated Development Environment (IDE) released by the Carnegie Mellon University.
[5] Game Maker is an IDE for creating videogames, developed by Mark Overmars and YoYo Games.
[6] Greenfoot is an interactive Java development environment created by the University of Kent.
[7] Kodu is a visual programming language specifically designed for creating videogames, developed by the Microsoft Research department.

Apple's HyperCard[8] [77], so as to make learners start as users and then become modifiers and creators of computational artifacts [46]. Finally, educational robotics kits and hardware platforms such as Arduino [58], GoGo Board [72], and Micro:bit [90] also aim at showing children some of CT's applications to real world problems. Nevertheless, in most introductory CS courses, after some programming experiences with visual and tangible media students can come into contact with high-level mainstream languages such as Java and Python.

In consideration of recommendations from the academic community to encourage girls' interest for computing, several tools were created aiming at providing more opportunities for exposure to CT as well as engaging girls. There exist, in fact, *computational craft* kits such as LilyPad Arduino [58] that allow to combine computing and electronics with manual activities such as sewing and drawing. However, one of the most complete educational tools is MIT's App Inventor [83], a visual development environment than enables to build interesting Android applications by using Scratch-like graphical code blocks. More gender neutral if compared to other tools, it possesses both qualities that one might expect from a software of this kind: it permits to easily build one's own apps (low floor), while engaging, at the same time, with sophisticated CT concepts such as procedural thinking, including iterative, recursive, conditional and logical thinking, data abstraction, task decomposition and debugging (high ceiling).

Video gaming is also becoming more and more popular as a way for developing many 21st-century skills among children [36], nevertheless the possibility to exploit it in the teaching of CT has been quite underestimated in recent years. In 2011, an attempt in this direction has been made by Holbert and Wilensky [22] with FormulaT, a prototype video game conceived as a platform where kids can easily learn principles of kinematics while putting into practice *systematic computational strategies*. FormulaT is based on NetLogo, an agent-based coding environment where agents can take different forms: *turtles*, *patches*, *links* and the *observer* are provided [85]. Indeed, it has been proved that students are more engaged by CS courses when they are asked to abstract pertinent behaviors and assign them to agents, apply precise rules, and evaluate their work by performing simulations. Paulo Blikstein [5], assistant professor at the Stanford University Graduate School of Education, also took a leaf out of NetLogo's computational models for his secondary-level classes. However, agent-based programming remains barely used in modern CT research.

Nonetheless, not all computational tools used today focus on the various components of CT in the same way. For example, Kafai et al. [30] reported in a research paper from 2008 that young students who learn programming with Scratch are indeed in a position to understand and apply

[8] HyperCard is a programming tool designed for Apple Macintosh and Apple IIGS computers.

concepts such as data abstraction, conditional logic, and iterative and parallel thinking after a short introductory course. On the other hand, one of its limitations is that Scratch does not allow to experience even basic usages of functions and procedures. Providing a solution to this was in fact one of the main goals of its successor, Snap!. Future tools expressly built for promoting CT among children should not only include the features shared by today's most effective tools developing all skills now recognized as component elements of CT, but also follow the guidelines provided by recent research on human understanding of computation and children's approach to problem solving [37]. For example, with respect to children, an article by Rachel Keen [24] on the development of problem solving skills shows that they are largely driven by curiosity and a need for exploration. Moreover, they also seem to be highly sensitive to the efficiency of their actions, which leads them to often change their strategies in order to achieve more efficient behaviors in pursuit of goals.

With reference to this, the Interaction Design and Children (IDC) conference was founded and first run in 2002 by two researchers – Panos Markopoulos and Mathilde Bekker – from the Technical University of Eindhoven (Netherlands). Since its inception, the conference has been annually housed by leading universities across Europe and the United States.

It can be described as an interdisciplinary international community that focuses on the opportunities and challenges of leveraging technology for educational purposes, in order to enable children to take part in nurturing and empowering experiences as well as bring their voice and opinions into the learning process. The IDC conference sets as its mission to bring together designers, educators, and researchers from the cognitive sciences, learning sciences, human-computer interaction, and the various formal and informal educational settings, so as to explore new forms of technology, design and engaged learning among school-aged children. The annual conference incorporates papers, presentations, speakers, workshops, participatory design experiences and discussions that aim at creating better interactive experiences for children.

Every year, IDC accepts submissions from all over the world in the form of papers, demonstrations, workshops, course projects, and interactive child applications, with regard to the following topics:

- Innovative interactive technologies designed for children;
- Theoretically motivated arguments concerning interaction design and children;
- Empirical studies dealing with the interaction between children and technology;
- Methods and techniques supporting interaction design and children;
- Studies dealing with the effects of technology on children's lives and cognitive development;
- Analytical studies in the field of child-computer interaction and interaction design;

- Research in the field of constructive design for and with children;
- Studies examining children's involvement in the design process;
- Articles dealing with future vision, investigating trends and directions for the sector [78].

Finally, although CT researchers are currently active in a huge variety of environments and contexts, there still remain several unexplored areas, particularly with regard to the possibilities to create computer-made objects and the positive effects that activities of this kind would have on children's cognitive development. For this reason, numerous initiatives arose with the aim of promoting the construction of tangible computational artifacts, as well as informal *hacker* applications addressed to kids, and ubiquitous smartphones, all exciting opportunities made possible by today's technology. Among these, some of the most popular projects are the following:

- fab labs (*fabrication laboratories*) [32], small-scale workshops offering (personal) digital fabrication;
- makerspaces [93], collaborative work spaces inside schools, libraries or separate public/private facilities for making, learning, exploring and sharing that use both high tech and no tech tools;
- Maker Faire [67], a family-friendly event that celebrates the DIY (Do It Yourself) culture in the high tech field, which constitutes the basis of the *makers* movement;
- Instructables [79], a website specializing in user-created and uploaded DIY projects, which other users can comment on and rate for quality.

4.3.2 Assessing CT

Assessment also plays a key role in the creation of successful CT curricula. In fact, without a set of validated measures enabling teachers to assess how much students have assimilated, it would be difficult to give an effective evaluation of any course incorporating CT.

Recent studies such as Werner et al. [48] *Fairy Assessment* (2012) suggest to either exploit preexisting artifacts or applications created by students themselves, in order to evaluate their understanding of concepts such as abstraction, algorithmic thinking and conditional logic, as well as how they make use of this knowledge for problem solving. Task breakdown, debugging, and reverse engineering are also being judged as appealing evaluation parameters.

For example, Fields et al. [13], in their *Debuggems* article (2012), opted for debugging, assessing students' engineering and coding competencies by evaluating their ability to correct predesigned defective e-textile programs. In particular, they analyzed high school students'

collaborative engagement with a series of isomorphic deconstruction kits (debuggems) developed to assess their learning of coding, circuit design and creation (through sewing) in e-textiles by using LilyPad Arduino. The research was conducted by videotaping ten students collaborating in pairs as they worked to turn on LEDs in a project strategically designed with problems in poor crafting, non-functional circuitry design and insufficient coding. On the other hand, Basawapatna et al. [18], in their paper on *Automatic Recognition of Computational Thinking* (2010) tried to mainly focus on reverse engineering, namely the ability to take something apart to reproduce it, in the attempt to give an answer to questions like this: *Now that the student can program Space Invaders, can the student program a science simulation?* More specifically, departing from the fact that current evaluation methods do not always make it clear what CT concepts students have actually learned, they attempted to develop a visual semantic evaluation tool – called the Computational Thinking Pattern (CTP) graph – for student-created games and simulations that goes towards depicting the CT concepts implemented by the students.

4.4 Concluding remarks and suggestions for future enquiries

The increasingly important role of computing and algorithms in particular has become universally recognized in recent years. For example, search engines allows us to retrieve massive amounts of data. Our preferences are mapped by recommendation algorithms, which let us come into contact with new bits of culture we are likely to be interested in. Our interactions on social media are also managed by specific algorithms, which may put in evidence the news related to a specific friend of ours while excluding what concerns another. All these algorithms together not only allow us to retrieve information, but also provide a means to discover what is to be known and how to know it. In such a sense, they have become a key logic governing the information flows on which we depend.

What clearly emerges from our analysis is that most recent research on CT has mainly focused on definitional matters and on creating tools aimed at fostering the development of computational skills. Some notable steps forward have been taken with regard to the definition of curricula including CS and CT basics, and the relevant assessment strategies. However, there still exist remarkable inadequacies that demand a more in-depth analysis and further empirical enquiries. In fact, the teaching of CS and CT is still subject to different interpretations, for what concerns both the topics to be covered in the relevant courses and the choice of the assessment criteria.

After all, Wing herself asserted that *an application of the science of learning research in designing grade- and age-appropriate curricula for computational thinking is necessary to*

maximize its impact on and significance for K-12[9] *students* [35]. Nonetheless, except for some recent studies, such as the ones carried out by Black et al. [11] and Berland and Lee [3], few other scholars have actually taken into consideration the results from contemporary research in the field of learning sciences, such as the notions of sociocultural and situated learning[10], or distributed and embodied cognition[11]. In the 1980s the study of cognitive processes typical of children and inexperienced programmers drew the attention of many influential researchers, such as Clement et al. [26], who analyzed the development of thinking skills; Pea et al. [40], who dealt with debugging; Clements and Gullo [8], who focused on transfer issues; Carver and Klahr [25] who studied the construction of effective scaffolds to facilitate transfer; just to mention a few. This abundant literature could prove to be quite useful for contemporary CT research.

The idea of using computing as an instrument for teaching other disciplines, interweaving the introduction of CT at the primary and secondary level with the application of problem solving skills in different domains, is underinvestigated as well. Nonetheless, there exist several past studies which demonstrated children's ability to learn mathematics [19] and science [23] through the development of LOGO programs.

Extensive research was also conducted with respect to the most common problems beginner CS students go through during their first programming experiences, which are not related to syntactical issues, in the attempt to discover whether there exist specific obstacles that arise in the path of educating children to CT, and, if that is the case, what are these hurdles and how they should be faced. Current studies on the teaching CT could take great advantage of past research in this field, too.

The development of strategies aimed at contrasting students' erroneous preconceptions concerning CS and CT is also a largely untouched field [34]. In fact, these stereotypes (such as the association of men and science, which is one of the reasons for gender imbalance in the field of IT, or the association of CS with *nerds*, *geeks* or *hackers*, which can give it a negative image) play a crucial role in the learning process and should be taken into account if we aspire to provide both

[9] With reference to the United States, K-12 comprises the sum of primary and secondary education for publicly-supported grades prior to college [81].

[10] The Sociocultural Learning theory was introduced by the Russian psychologist Leo Vygotsky, and is based on the idea that a learner's environment plays a pivotal role in his/her learning development [80].
The Situated Learning theory, instead, was first proposed by Jean Lave and Etienne Wenger, and *takes as its focus the relationship between learning and the social situation in which it occurs* [28].

[11] The Distributed Cognition (DCog) theory, crafted by Edwin Hutchins, posits that cognition and knowledge are not confined to an individual; rather, they are distributed across objects, individuals, artifacts, and tools in the environment [69].
The Embodied Cognition theory holds that an agent's cognition is strongly influenced by aspects of an agent's body beyond the brain itself [51].

boys and girls with learning experiences that aim at nurturing CT skills. Recent studies on students' attitudes towards computing actually paved the way for a better management of this.

Of course, there is still a long way to go to help children develop a sound theoretical and practical understanding of computing. Furthermore, we will not be in a position to make any serious attempt to introduce CT in school curricula before answering some fundamental questions such as, for example: what can we expect children to know or be able to do after taking part in a course designed to develop CT competencies? And then: how can we assess the skills they have acquired? In our opinion, the time has come to close the gaps and widen the academic discourse on CT education.

Chapter 5

Proposal for a Coding Course

5.1 Introduction

In the course of this work we have firstly introduced the concept of computational thinking and described the most significant initiatives worldwide aimed at promoting its learning in primary and secondary schools. Subsequently, we have analyzed some of the most common didactic tools used for this purpose, and provided an overview of the state of the art with respect to research on computing didactics.

As a conclusion, I would like to present a personal proposal for an introductory coding course of four classes lasting about two hours each, addressed to non Computer Science students. My idea is not to propose a course specifically designed for primary school children, but a coherent set of activities adaptable to various situations, such as a class composed by high school or college students. This is why I would not choose to focus on a child-specific tool such as Scratch or the Blockly library, but would rather opt for a more advanced level software, such as Snap! or Stencyl. Between the latter, Snap! could be a more suitable option in my opinion, because of the possibility it provides to encode programs to mainstream languages such as Python, JavaScript, C, etc. In what follows, I am going to describe how a course of this kind might be organized, illustrating the objectives and contents of each class, as well as the teaching methods adopted. The table in Figure 5.1 shows a prospectus of the course.

	Objectives	Contents	Teaching methods	Tools
Lesson 1	Understand the role of programs and how they interact with computers.	Notions of program and computer; programming languages and machine code; compilation and interpretation; low-level and high-level programming.	Frontal lecturing.	None
Lesson 2	Understand the mode of operation of Snap!; create a program that draws a square.	Installation procedures; user interface; block categories.	Frontal lecturing, practice exercise.	Snap!
Lesson 3	Understand the usefulness of customized blocks; create small programs that include at least one customized block.	Creation and usage of customized blocks; examples related to Command, Reporter and Predicate blocks.	Practice exercise.	Snap!
Lesson 4	Understand the usefulness of customized blocks with inputs; create small programs that include at least one customized block requiring input parameters.	Creation and usage of customized blocks with inputs; examples related to Command, Reporter and Predicate blocks.	Practice exercise.	Snap!

Figure 5.1. Summary table of my course proposal

5.2 Lesson 1

I would find it useful to devote the first class to a general introduction to programming. In my opinion, students should first understand what programs are and how they interact with computers. In particular, I would highlight the difference between a computer, which can be described as a tool for solving problems with data, and a program, which is a sequence of instructions telling a computer how to do a given task. Subsequently, I could start illustrating the basic structure of a computer, built as a collection of switches, where the ON state corresponds to the value 1, while the OFF state corresponds to the value 0.

At this point, students could better understand the importance of programming languages as tools that allow to express computer-readable binary sequences of 1s and 0s (*machine code*) in a human-readable format. With regards to this, the concepts of *compilation* and *interpretation* can also be introduced, putting in evidence both their common objective, that is to *translate* an

algorithm written in a given programming language into machine code, and their different approaches.

Compilation, in fact, consists of converting the original program into a binary source file that will be permanently stored. Interpretation, instead, consists of modifying the human-readable content into machine code when the program is executed. Pros and cons of both types of software development should also be illustrated. For example, compiled programs are generally faster to run, because the computer only needs to execute the previously translated instructions. On the other hand, they are often slower to develop, since interpreted languages are simpler and there is no need to compile the whole program each time you want to test a new feature.

To conclude the introductory lesson, I would find it useful to illustrate the main differences between low-level and high-level programming. The former, in fact, is closer to machine code, while the latter resembles more closely natural languages. The assembly language should also be mentioned as the lowest level existing language, that is just a direct translation of the binary instructions executed by the computer. Its dependence on the processor architecture is also relevant. In fact, each machine code instruction corresponds to exactly one in assembly language. Therefore, every kind of processor architecture relates to a specific machine code as well as a specific assembly language.

5.3 Lesson 2

In the second class I would present the Snap! software. After a quick overview of installation procedures, I would start out by describing the user interface, illustrated in Figure 5.1, as made up of three columns, the first of which contains a list of code blocks, the middle one is where these blocks are to be dragged into and put in order so as to command the actions of the default sprite (the small arrow inside the big white rectangular area), and the last one, including the big white rectangle, is where we are going to see what our blocks actually do.

Afterward, I would start examining the different block categories provided by Snap!. The Motion section is probably the most important for our purposes, as it includes all those commands that allow to move the sprite around the screen. However, the other sections (Looks, Sound, Pen, Control, Sensing, Operators, and Variables) also deserve to be at least briefly illustrated.

The Control section, for example, includes the *when green flag clicked* block, which needs to be set at the beginning of any program to allow the user to start the application by clicking on the green flag above the white rectangle. After illustrating this command, I would go and explain the

function of two basic motion instructions, namely *move … steps* and *go to x: … y: …*, showing the students how changes on the relevant numeric values modify the sprite's behavior.

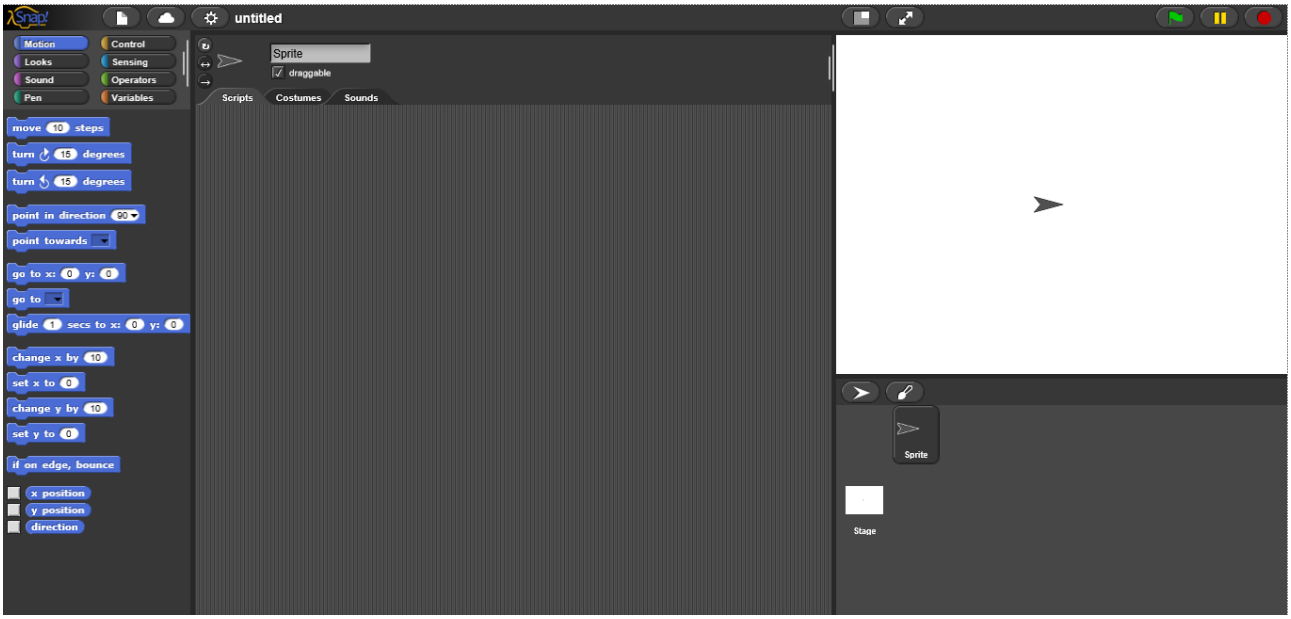

Figure 5.1. Snap! user interface at startup

At this point, I could set as the goal of my program to make the sprite draw a square. This requires me to introduce two commands from the Pen section, namely *pen down* and *set pen color to …*, respectively needed to make the sprite draw a line and to set its color. Then, I would show what happens if we attach the command *turn right 90 degrees* to the previous blocks, because squares have right angles between the sides, and click the green flag repeatedly. What we obtain is illustrated in Figure 5.2.

Of course, the picture we obtain by running this script does not represent a square, thus we need to add some further instructions. In particular, we need to include the *clear* and *pen up* blocks from the Pen section, which respectively clear the screen and make the sprite not to draw any line at the beginning of a new program execution. Moreover, we want the sprite to be facing in a particular direction when drawing a line, that is why we also need the motion command *point in direction 90*, which makes the sprite to point to the right. Lastly, we need to repeat four times a command sequence such as *move 100 steps*, *turn right 90 degrees*, once for each square side. Fortunately, this can be simplified by inserting this sequence inside the *repeat …* block, which can be found in the Control section. The complete script and the relevant output are illustrated in Figure 5.3.

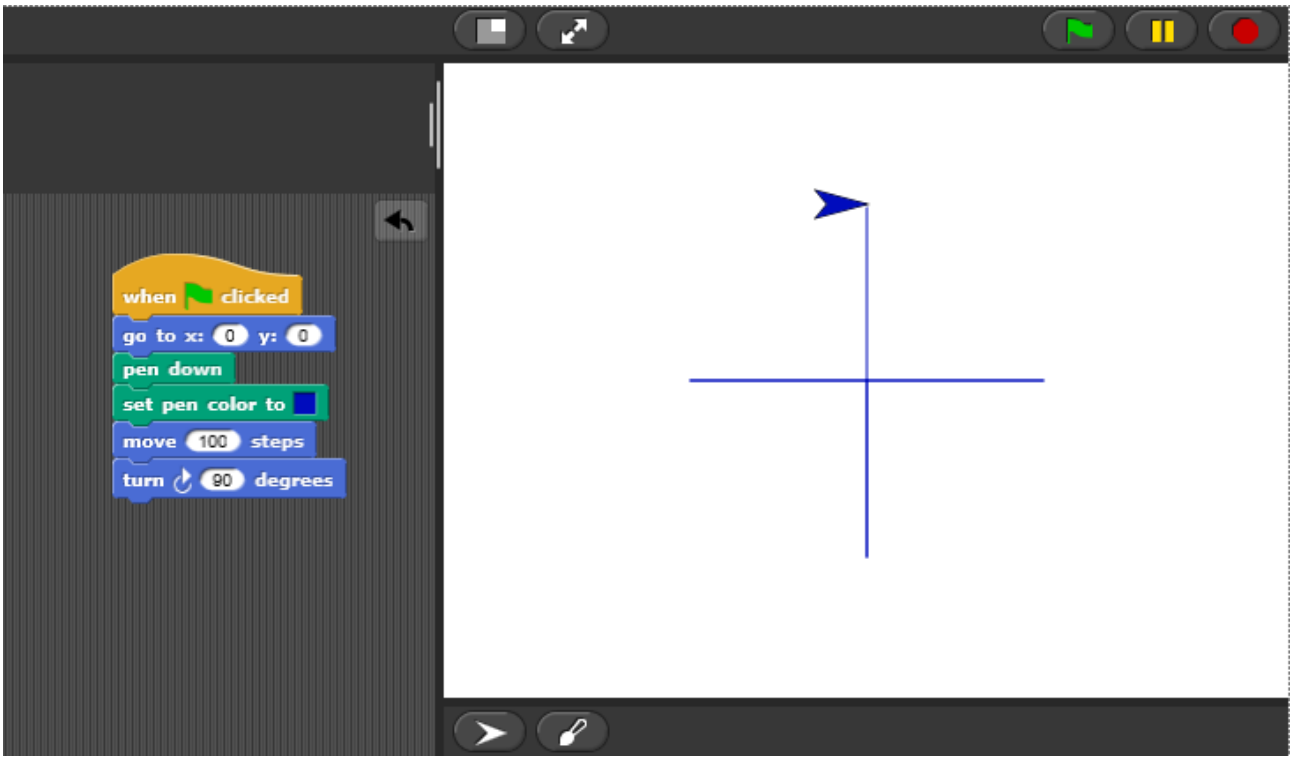

Figure 5.2. Snap! script (example 1)

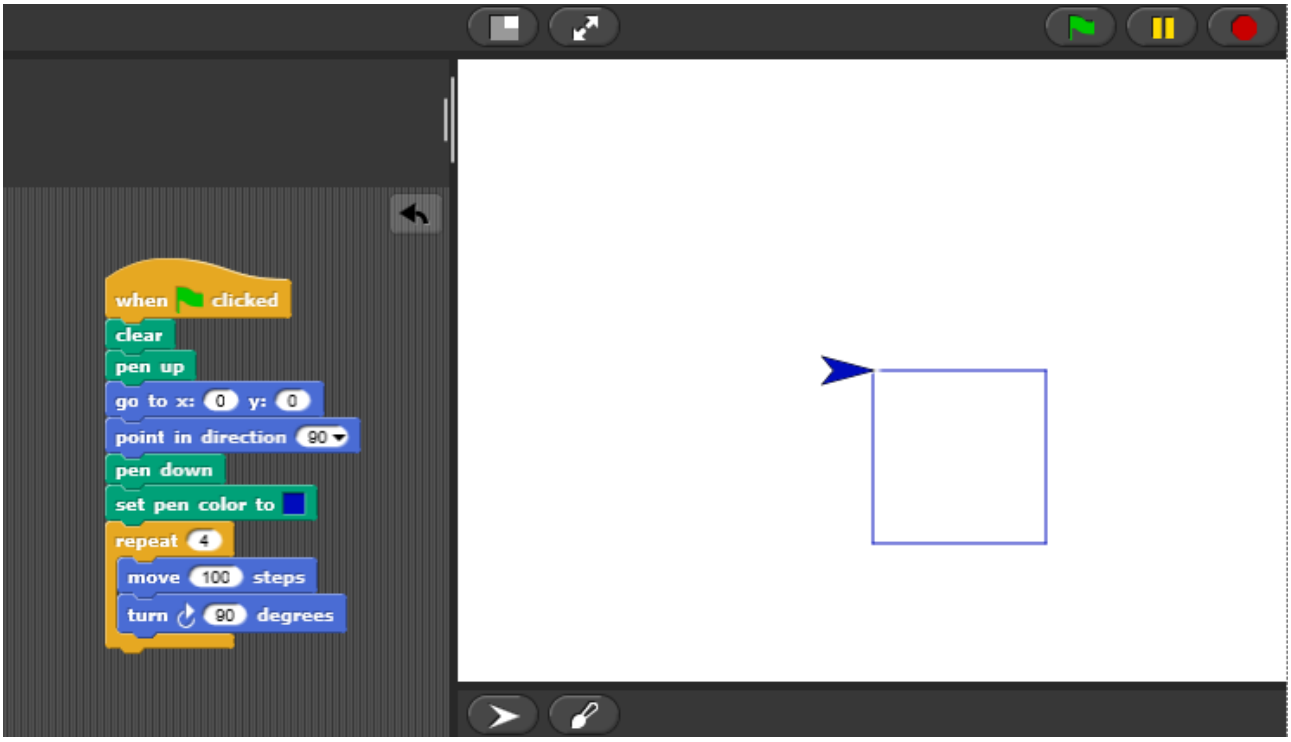

Figure 5.3. Snap! script (example 2)

5.4 Lesson 3

The third class should be devoted, in my opinion, to an interesting Snap! feature, that is the possibility to create and use further blocks. To illustrate this feature, we need to examine the dialog window we are presented with when clicking the *make a block* button from the Variables section (Figure 5.4).

We can see that there are four choices to make. First, into which of the block menus should our block go, or, in other words, which category should it belong to; second, what is its name; third, what shape block it is, and with respect to this, three options are provided: we can either create a jigsaw shaped command block, an oval reporter block, or a hexagonal predicate block; fourth, whether this block should be available to all sprites or just to the current sprite. If we want to build a block that draws a square, we could select the Motion category, name our block *square*, assign to it the shape of a command block, and make it available for all sprites.

Figure 5.4. *Make a block* dialog window

Once we have made all four of these choices, the so called block editor will open up, which is a little scripting area where we need to specify what Snap! should do when our block is used. Since we want it to draw a square, we simply have to drag into the block editor the basic instructions needed to perform this task. As illustrated in Figure 5.5, I have chosen not to include commands such as *go to x: ... y: ...*, to set the sprite's position, or a *pen down* command, even though I am sure

I will need it when using the *square* block, because I find it more useful to keep the block definition as simple as possible, and put these scene setting commands in the script using *square*.

Figure 5.6 shows a possible script that uses this new block, now appearing at the bottom of the Motion menu. In particular, it draws a bunch of squares making the sprite turn right a little bit (36 degrees) in between. The relevant result is also illustrated in the figure. As a further example, I have also used the *square* block in a slightly more complicated script, whose code and result are presented in Figure 5.7. In this case, at each repetition the sprite puts the pen down, draws a square, picks the pen up and then moves a little bit (20 steps) before turning right. These two examples make it easy to understand why it is generally useful to avoid making strong assumptions inside the block definition. In fact, I did not want to set the position of the square or the moment in which the sprite should start drawing, identified by the *pen down* command. This allows me to use the *square* block multiple times in multiple positions.

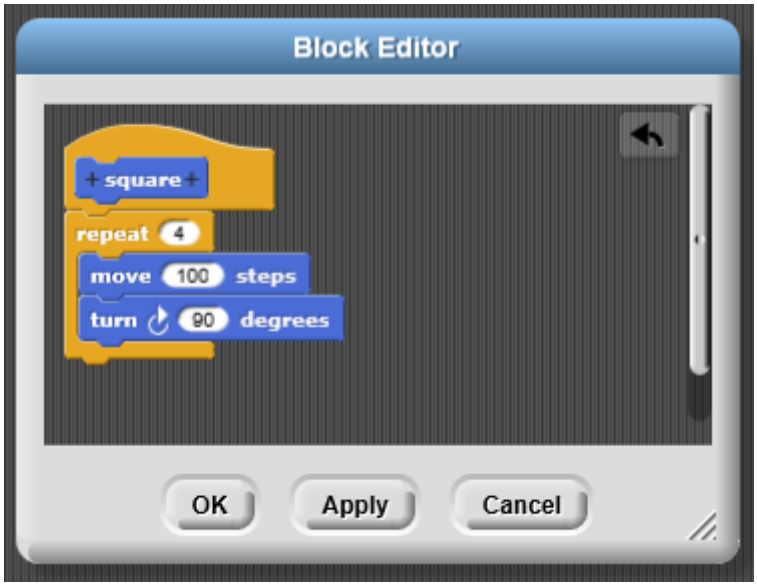

Figure 5.5. Block editor script

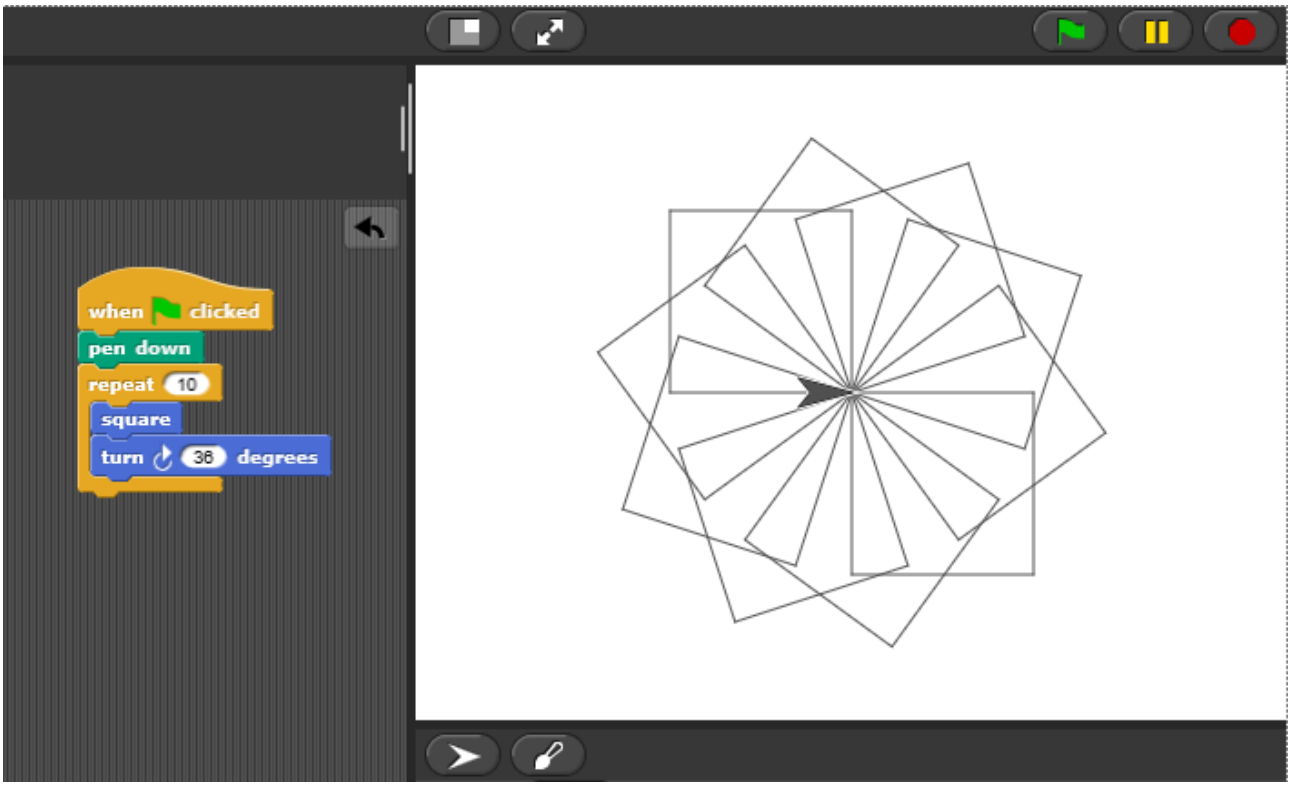

Figure 5.6. Snap! script (example 3)

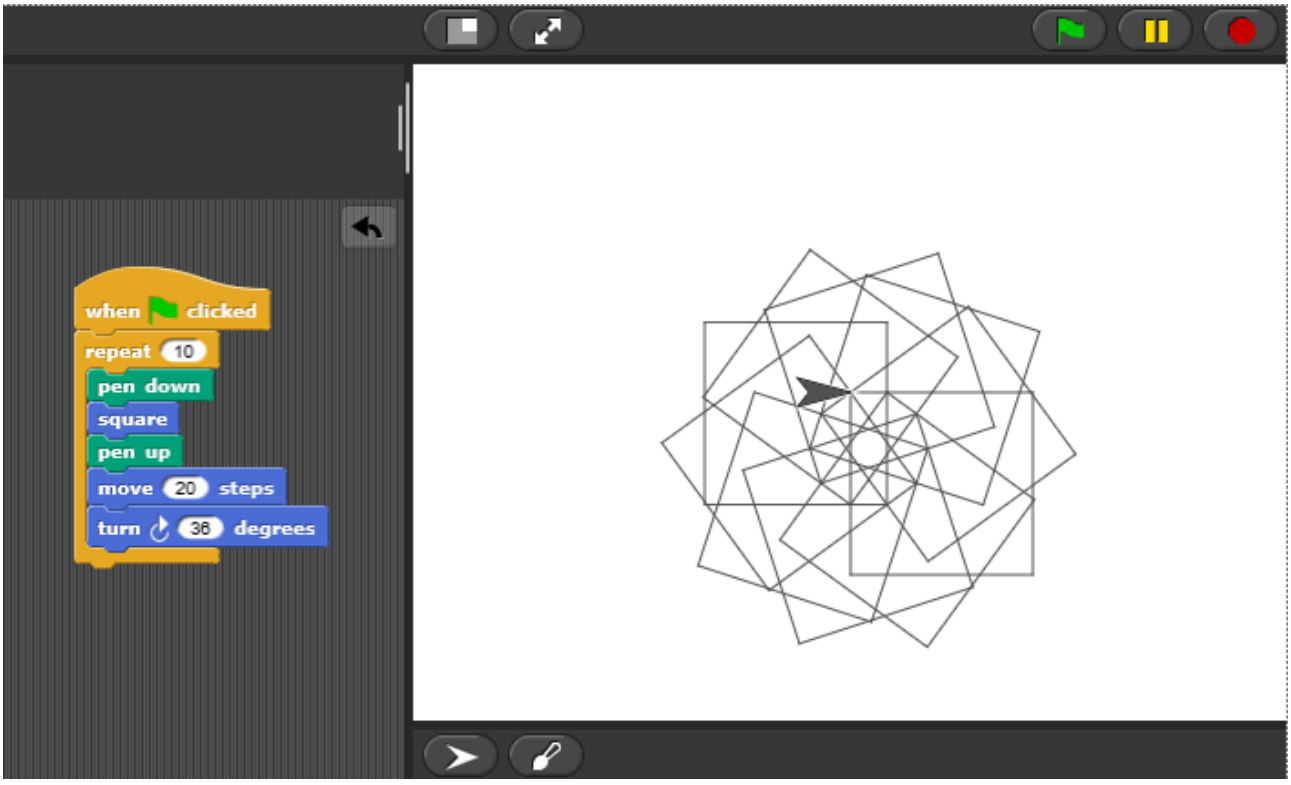

Figure 5.7. Snap! script (example 4)

Of course, it could be useful to expand this class with further examples. In particular, the creation and use of reporter and predicate blocks might be illustrated, respectively needed to report a specified value in the case of reporter blocks, and to tell whether a given condition is satisfied or not in the case of predicate blocks.

5.5 Lesson 4

The fourth and last class would be devoted to the creation of customized blocks with inputs. In fact, if we consider the previously built *square* block, we can see that, unlike the other blocks in the Motion menu, it does not include any input field where to specify one or more features of the square we want to draw, such as its size or color.

To fix this, we need to modify the relevant block definition (Figure 5.5), which can be accessed by right-clicking the block itself and then selecting the *edit* option. First, we notice that there are two plus signs, one for each side of the word *square*, that light up if we hover over them. Clicking one of these plus signs tells Snap! that we would like to add an input field in that position, before or after the command word. Now, we want to add an input field to specify the square size. To do this, we need to click one of the two plus signs. As a result, a dialog window entitled *Create input name* should open up, where we simply enter *size* as an input name, as shown in Figure 5.8.

At this point, a variable called *size* should appear in the hat-shaped block of the *square* definition (Figure 5.9). Moreover, if we click *Apply*, we will see that every instance of the *square* block changes, so that it now has an input field. Nevertheless, the input variable is not going to do anything unless we use it in some way.

Now, we would like the *size* input to determine the size of the square we are going to draw. Of course, we cannot change the number of repetitions, since a square always has four sides, and cannot even change the turning angle, which has to be always 90 degrees. Differently, we need to pick up the *size* variable and drag it down to the *move ... steps* block, so that, instead of having a command that says *move 100 steps*, it now says *move "size" steps* (Figure 5.10).

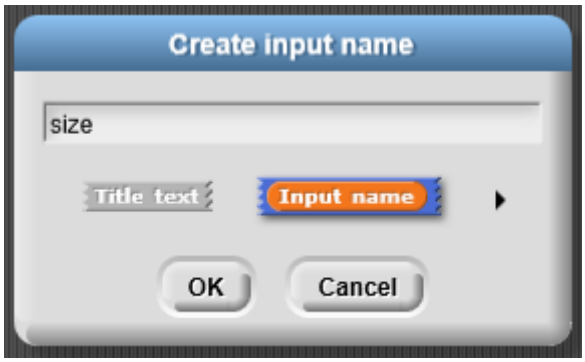

Figure 5.8. *Create input name* dialog window

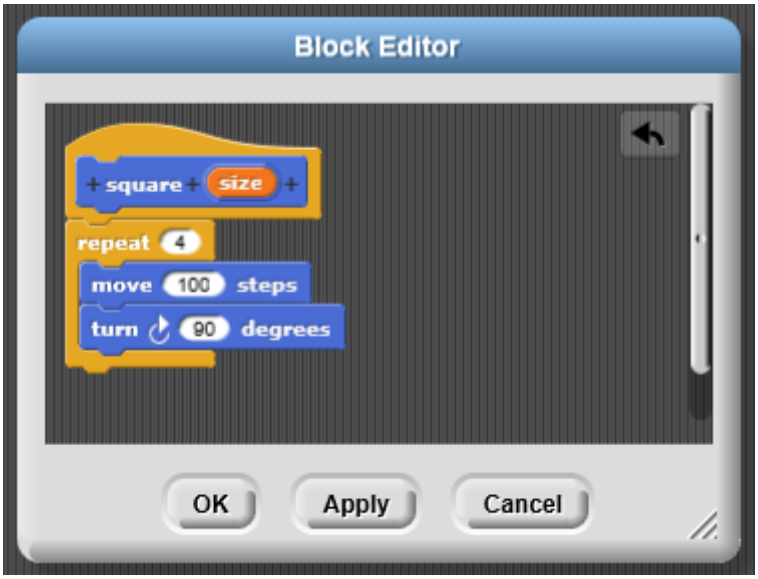

Figure 5.9. Block editor script with input variable (1)

So, if we now consider a basic square drawing script, such as the one proposed in Figure 5.11, and enter 50 in the *square* block, what we will obtain is a smaller square. On the contrary, if we try and enter 150, we will get a bigger square. Figure 5.11 illustrates what we get as a result for the values 50, 100, 150.

Similarly to the previous class, further examples might be added, concerning the use of input parameters with reference to reporter and predicate blocks.

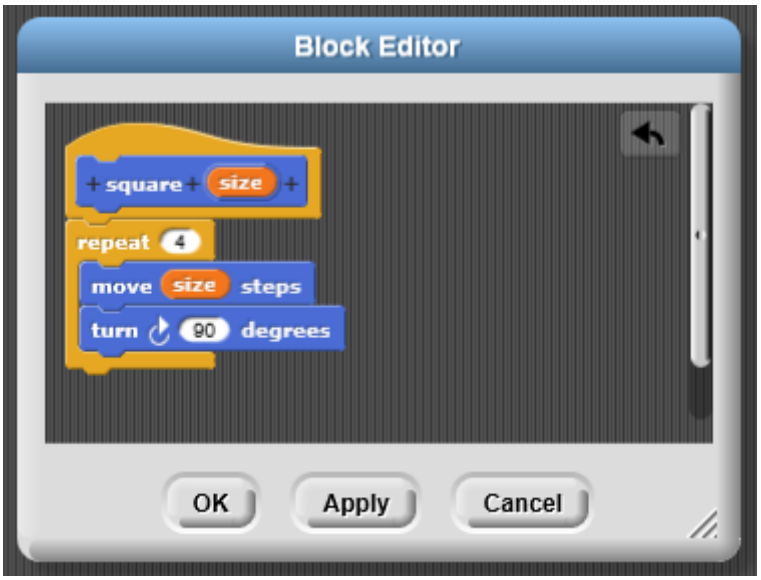

Figure 5.10. Block editor script with input variable (2)

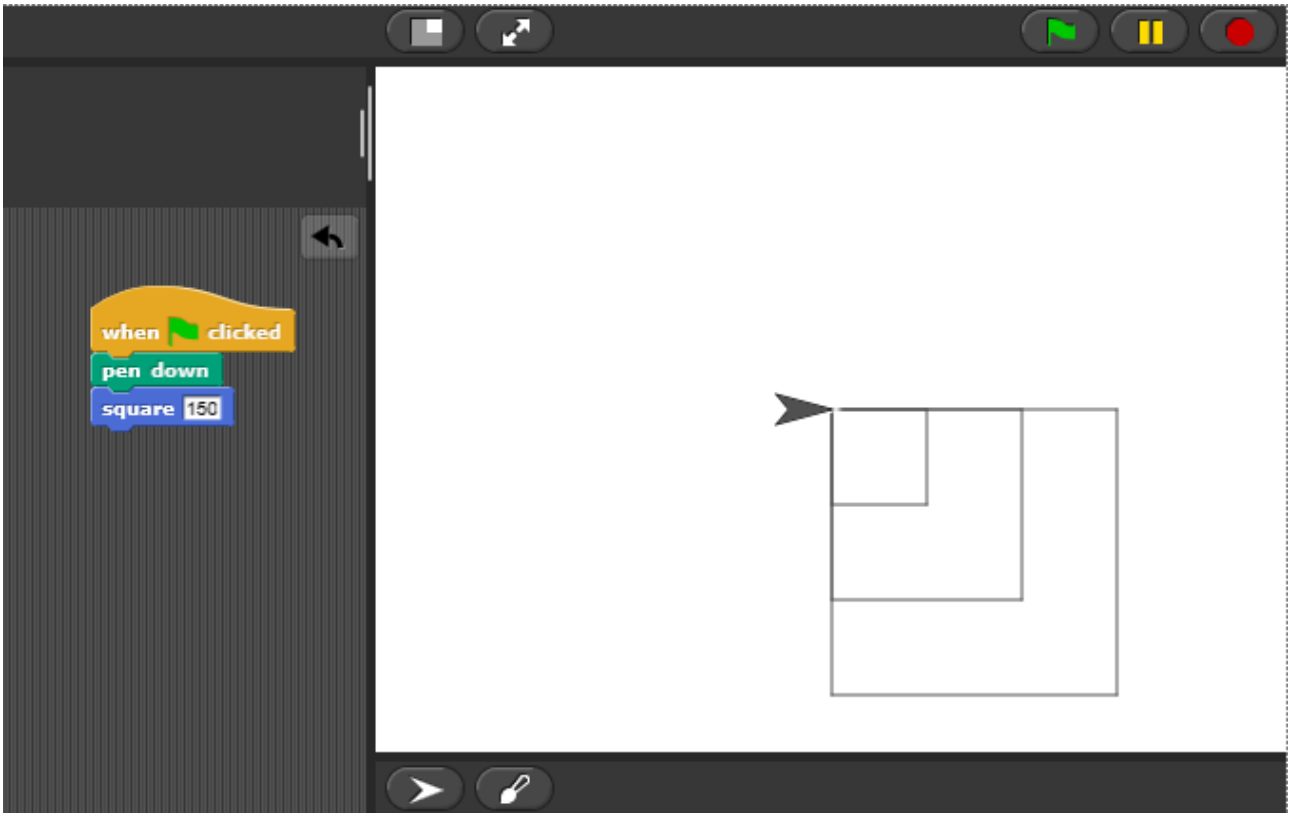

Figure 5.11. Snap! script (example 5)

In presenting these four coding classes, my purpose was neither to design a complete nor adequately exhaustive course, but I wanted to propose a set of modules which could be used with a certain degree of flexibility to provide high school or college students with some basic cognitive tools that are needed to start an effective and structured learning of computer science and

programming in particular. Such a choice has been motivated, on the one hand, by a need for pragmatism, in the attempt to provide a personal contribution starting from my previous analysis of the existing situation with respect to CS education; on the other hand, it has been inspired by the teaching experience acquired in the course of an internship I have recently undertaken in a public school.

To summarize, I believed it could be useful to organize this breakthrough course as follows. The first class should be devoted to an preliminary introduction to programming, focused on presenting the concepts of program and programming language, the differences between high level and low level programming, and the main properties of the two possible implementation approaches (compilation versus interpretation). The second class should focus on illustrating the main features of the used coding tool, which, in the proposed scenario, could be Snap!. The third class should concentrate on how to create and use procedures, represented by customized blocks in Snap!, or something similar to them. The fourth and last class should be devoted to how to create simple programs that are able to receive some input data, that is why I would opt for illustrating the use of customized blocks with inputs.

Conclusion

We have started this work by introducing the notion of computational thinking and presenting an overview of the most significant literature around it. Secondly we have illustrated the most significant initiatives worldwide aimed at promoting its learning in the context of primary and secondary education. Thirdly, we have described some of the most common block programming languages used for didactical purposes. Fourthly, we have provided an overview of the state of the art with respect to research on computing didactics and design of interactive applications specifically addressed to children. Lastly, we have presented a proposal for an introductory coding course designed for non-Computer Science students.

Although courses of this kind already exist in Italy and abroad, I believe that Computer Science education is currently facing some challenges that other disciplines, such as engineering, were facing some years ago. Indeed, several initiatives – such as Primary Engineer and Young Engineers (UK) – have been launched in the past few years to promote the integration of engineering education at primary level. The main reason in favor of these initiatives has been its practical nature, through which it is able to engage children in mathematics and science. Similarly, Computer Science is struggling to find its place in the context of primary and secondary education, and to learn to conduct educational research that can be considered robust and credible. Indeed, it is clear that research on Computer Science education is not up to par with educational research in other disciplines. This is probably due to the fact that articles and studies in this field are written by computer scientists, whose approach to research takes a notably different perspective than an educational scientist might. The foundations of Computer Science lie in mathematics and logic, which are distinct from educational sciences with respect to the degree of certainty that is the result of research. Moreover, we have to admit that several aspects of Computer Science extend into the realm of natural sciences (for example, Artificial Intelligence includes applications from physics, biology, and even psychology), but these forays into educational sciences do not effectively prepare computer scientists for research in this field. Social sciences are definitely much more complex than

pure logic, as they require research to rely on theoretical frameworks and provide links to previous studies.

In the next few years, I believe that the need for a close collaboration between Computer Science experts and educational or social science researchers will be felt more and more. If this comes true, it will be possible to create effective Computer Science curricula, able to meet the needs of primary and secondary level education, but also consistent with what most companies of the IT sector require from schools.

References

Literature

[1] A. V. Aho. “Computation and computational thinking”. In: *Computer Journal* (2012).

[2] V. Barr, C. Stephenson. “Bringing computational thinking to K-12: What is involved and what is the role of the computer science education community?”. In: *ACM Inroads* (2011).

[3] M. Berland, V. Lee. “Collaborative strategic board games as a site for distributed computational thinking”. In: *International Journal of Game-Based Learning* (2011).

[4] C. M. Bishop. *Pattern Recognition and Machine Learning.* Springer, 2006.

[5] P. Blikstein. “Connecting the science classroom and tangible interfaces: the bifocal modeling framework”. In: *Proceedings of the 9th International Conference of the Learning Sciences*. Chicago, IL, 2010.

[6] P. Blikstein. *Seymour Papert’s Legacy: Thinking about learning, and learning about thinking*. 2013. URL: https://tltl.stanford.edu/content/seymour-papert-s-legacy-thinking-about-learning-and-learning-about-thinking.

[7] E. Cassese. *E se il Coding Fosse la Nuova Lingua Straniera? – Educazione Globale.* 2014. URL: http://www.educazioneglobale.com/2014/02/e-se-il-coding-fosse-la-nuova-lingua-straniera.

[8] D. H. Clements, D. F. Gullo. “Effects of computer programming on young children’s cognitions”. In: *Journal of Educational Psychology* (1984).

[9] P. Denning, P. Freeman. “Computing’s paradigm”. In: *Communications of the ACM* (2009).

[10] A. A. diSessa. *Changing Minds: Computers, learning, and literacy.* Cambridge: MIT Press, 2000.

[11] C. L. Fadjo, M. Lu, J. B. Black. *Instructional embodiment and video game programming in an after school program.* Paper presented at the World Conference on Educational Multimedia, Hypermedia & Telecommunications, Chesapeake, VA (June, 2009).

[12] C. Felker. *Maybe Not Everybody Should Learn to Code: A software engineer's take on the new education call to arms.* 2013. URL: http://www.slate.com/articles/technology/future_tense/2013/08/everybody_does_not_need_to_learn_to_code.html.

[13] D. A. Fields, K. A. Searle, Y. B. Kafai, H. S. Min. "Debuggems to assess student learning in e-textiles". In: *Proceedings of the 43rd SIGCSE Technical Symposium on Computer Science Education.* New York, NY: ACM Press, 2012.

[14] H. Giest. *Zur Didaktik des Sachunterrichts – Aktuelle Probleme, Fragen und Antworten.* Universitätsverlag Potsdam, 2009.

[15] S. Grover, R. Pea. "Computational thinking in K-12: A review of the state of the field". In: *Educational Researcher* (2013).

[16] M. Guzdial. *Learner-Centered Design of Computing Education: Research on computing for everyone. Synthesis lectures on human-centered informatics.* Morgan & Claypool Publishers, 2015.

[17] M. Guzdial. "Paving the way for computational thinking". In: *Communications of the ACM* (August 2008).

[18] K. Han Koh, A. Basawapatna, V. Bennet, A. Repenning. "Towards the automatic recognition of computational thinking for adaptive visual language learning". In: *Proceedings of the 2010 Conference on Visual Languages and Human Centric Computing (VL/HCC 2010).* Madrid, Spain: IEEE Computer (2010).

[19] I. Harel, S. Papert. "Software design as a learning environment". In: *Interactive Learning Environments* (1990).

[20] P. B. Henderson, T. J. Cortina, O. Hazzan, J. M. Wing. "Computational thinking". In: *Proceedings of the 38th ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '07).* New York, NY: ACM Press, 2007.

[21] M. B. Hesse. *Models and Analogies in Science.* Sheed & Ward Ltd., 1963.

[22] N. R. Holbert, U. Wilensky. *Racing Games for Exploring Kinematics: A computational thinking approach.* Paper presented at the annual meeting of the American Educational Research Association, New Orleans, LA. April, 2011.

[23] Y. B. Kafai, C. C. Ching, S. Marshall. "Children as designers of educational multimedia software". In: *Computers & Education* (1997).

[24] R. Keen. "The development of problem solving in young children: A critical cognitive skill". In: *Annual Review of Psychology* (2011).

[25] D. Klahr, S. M. Carver. "Cognitive objectives in a LOGO debugging curriculum: Instruction, learning, and transfer". In: *Cognitive Psychology* (1988).

[26] D. M. Kurland, R. D. Pea, C. Clement, R. Mawby. "A study of the development of programming ability and thinking skills in high school students". In: *Journal of Educational Computing Research* (1986).

[27] A. Lamb, L. Johnson. "Scratch: Computer programming for 21st century learners". In: *Teacher Librarian* (April, 2011).

[28] J. Lave, E. Wenger. *Situated Learning: Legitimate peripheral participation.* Cambridge University Press, 1991.

[29] C. E. Leiserson, C. Stein, R. Rivest, T. H. Cormen. *Introduction to Algorithms.* MIT Press, 1990.

[30] J. Maloney, K. Peppler, Y. B. Kafai, M. Resnick, N. Rusk. "Programming by choice: Urban youth learning programming with Scratch. In: *Proceedings of SIGCSE '08*. New York, NY: ACM Press, 2008.

[31] M. Marji. *Learn to Program with Scratch. A visual introduction to programming with games, art, science, and math.* No Starch Press, 2014.

[32] M. Menichinelli. *Business Models for Fab Labs.* URL: http://www.openp2pdesign.org/2011/fabbing/business-models-for-fab-labs.

[33] E. M. Mercier, B. Barron, K. M. O'Connor. "Images of self and others as computer users: The role of gender and experience". In: *Journal of Computer Assisted Learning* (2006).

[34] National Research Council. *Committee for the Workshops on Computational Thinking: Report of a workshop on the scope and nature of computational thinking.* Washington, DC: National Academies Press, 2010.

[35] National Research Council. *Committee for the Workshops on Computational Thinking: Report of a workshop on the pedagogical aspects of computational thinking.* Washington, DC: National Academies Press, 2011.

[36] National Research Council. *A Framework for K-12 Science Education: Practices, crosscutting concepts, and core ideas.* Washington, DC: National Academies Press, 2012.

[37] J. F. Pane, C. A. Ratanamahatana, B. A. Myers. "Studying the language and structure in non-programmers' solutions to programming problems." In: *International Journal of Human-Computer Studies* (2001).

[38] S. Papert. *Mindstorms: Children, computers, and powerful ideas.* New York, NY: Basic Books, 1980.

[39] S. Papert. "Situating constructionism". In: I. Harel & S. Papert (Eds.), *Constructionism.* Norwood, NJ: Ablex, 1991.

[40] R. D. Pea, E. Soloway, J. C. Spohrer. "The buggy path to the development of programming expertise". In: *Focus on Learning Problems in Mathematics* (1987).

[41] A. Repenning, D. Webb, A. Ioannidou. "Scalable game design and the development of a checklist for getting computational thinking into public schools". In: *Proceedings of the 41st ACM Technical Symposium on Computer Science Education.* New York, NY: ACM Press, 2010.

[42] M. Resnick. *Learn to Code, Code to Learn.* 2013. URL: http://web.media.mit.edu/~mres/papers/L2CC2L-handout.pdf.

[43] M. Resnick. *Let's Teach Kids to Code.* 2013. URL: http://www.ted.com/talks/mitch_resnick_let_s_teach_kids_to_code/transcript.

[44] Royal Society. *Shut Down or Restart: The way forward for computing in UK schools.* 2012. URL: https://royalsociety.org/~/media/education/computing-in-schools/2012-01-12-computing-in-schools.pdf.

[45] B. A. Sheil. "Teaching procedural literacy: Presentation abstract". In: *Proceedings of the ACM 1980 Annual Conference* (1980).

[46] C. Stephenson, J. Malyn-Smith. *Computational Thinking from a Dispositions Perspective.* 2016. URL: https://blog.google/topics/education/computational-thinking-dispositions-perspective.

[47] S. Watanabe. "Pattern recognition as conceptual morphogenesis". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (1980).

[48] L. Werner, J. Denner, S. Campe, D. C. Kawamoto. "The Fairy performance assessment: Measuring computational thinking in middle school". In: *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education (SIGCSE '12).* New York, NY: ACM Press, 2012.

[49] C. Wilson, M. Guzdial. "How to make progress in computing education". In: *Communications of the ACM* (2010).

[50] C. Wilson, L. A. Sudol, C. Stephenson, M. Stehlik. *Running on Empty: The failure to teach K-12 computer science in the digital age.* 2010. URL: http://runningonempty.acm.org/fullreport2.pdf.

[51] R. A. Wilson, L. Foglia. "Embodied cognition". In: *The Stanford Encyclopedia of Philosophy.* 2011.

[52] J. M. Wing. "A call to action: Look beyond the horizon". In: *IEEE Security & Privacy* (2003).

[53] J. M. Wing. "Computational thinking". In: *Communications of the ACM* (March, 2006).

[54] J. M. Wing. *Computational Thinking and Thinking About Computing*. 2008. URL: http://www.cs.cmu.edu/~wing/publications/Wing08a.pdf.

[55] J. M. Wing. "Research notebook: Computational thinking – What and why?" In: *The Link Magazine* (Spring, 2011). Carnegie Mellon University, Pittsburgh.

Online sources

[56] URL: www.blocklanguages.org.

[57] *Alice (software) – Wikipedia.* URL: https://en.wikipedia.org/wiki/Alice_(software).

[58] *Arduino – Home.* URL: https://www.arduino.cc.

[59] *BBC Bitesize – Home.* URL: http://www.bbc.co.uk/education.

[60] *BJC – Beauty and Joy of Computing.* URL: http://bjc.berkeley.edu.

[61] *Code Club | Home.* URL: https://www.codeclub.org.uk.

[62] *CodingGirls Roma-Usa | Fondazione Mondo Digitale.* URL: http://www.mondodigitale.org/it/cosa-facciamo/aree-intervento/pna/codinggirls-roma-usa.

[63] *Computational Thinking for Educators – Course.* URL: https://computationalthinkingcourse.withgoogle.com.

[64] *Computer Science Teacher: Programming with blocks.* URL: http://blog.acthompson.net/2012/12/programming-with-blocks.html.

[65] *Computer Science Unplugged.* URL: http://csunplugged.org.

[66] *Computing in the Core has moved – Code.org.* URL: https://code.org/computing-in-the-core.

[67] *Cosa è Maker Faire – Maker Faire Rome.* URL: http://www.makerfairerome.eu/it/cosa-e.

[68] *CS50 Syllabus.* URL: https://cdn.cs50.net/2015/x/references/syllabus/syllabus.html.

[69] *Distributed Cognition (DCog) – Learning Theories.* URL: https://www.learning-theories.com/distributed-cognition-dcog.html.

[70] *Exploring Computer Science.* URL: http://www.exploringcs.org.

[71] *GameMaker: Studio – Wikipedia.* URL: https://en.wikipedia.org/wiki/GameMaker:_Studio#cite_note-1.

[72] *GoGo Board*. URL: http://gogoboard.org.

[73] *Google CS4HS.* URL: https://www.cs4hs.com.

[74] *Google for Education: A solution built for teachers and students.* URL: https://edu.google.com.

[75] *Greenfoot.* URL: https://www.greenfoot.org/door.

[76] *H-FARM.* URL: http://www.h-farm.com.

[77] *HyperCard – Wikipedia.* URL: https://en.wikipedia.org/wiki/HyperCard.

[78] *IDC 2017 – ACM SIGCHI Interaction Design and Children.* URL: http://idc2017.stanford.edu.

[79] *Instructables – How to make anything.* URL: http://www.instructables.com.

[80] *Instructional Design Models and Theories: The Sociocultural Learning Theory: eLearning Industry*. URL: https://elearningindustry.com/sociocultural-learning-theory.

[81] *K-12 – Wikipedia.* URL: https://en.wikipedia.org/wiki/K%E2%80%9312.

[82] *Kodu | Home*. URL: https://www.kodugamelab.com.

[83] *MIT App Inventor | Explore MIT App Inventor*. URL: http://appinventor.mit.edu/explore.

[84] *Modularity definition and information.* URL: https://www.defit.org/modularity

[85] *NetLogo 6.0.1. User Manual: Programming Guide*. URL: http://ccl.northwestern.edu/netlogo/docs/programming.html.

[86] *News & Updates - MIT Media Lab.* URL: https://www.media.mit.edu.

[87] *Programmieren Lernen für Kinder: mit Spaß fit für die Zukunft.* URL: https://www-de.scoyo.com/eltern/kinder-und-medien/programmieren-lernen-kinder-fit-fuer-die-zukunft.

[88] *Scratch – Imagine, Program, Share*. URL: https://scratch.mit.edu.

[89] *Scratch (programming language).* URL: https://en.wikipedia.org/wiki/Scratch_(programming_language).

[90] *The Micro:bit Foundation is a global non-profit organisation making invention with technology fun for everyone!* URL: http://microbit.org.

[91] *The world's leading software development platform – GitHub*. URL: https://github.com.

[92] *U.S. Bureau of Labor Statistics.* URL: https://www.bls.gov/ooh.

[93] *What is a Makerspace? Is it a Hackerspace or a Makerspace?* URL: https://www.makerspaces.com/what-is-a-makerspace.