Overview of the Computer Programming Learning Environments for primary education

GEORGIOS FESSAKIS[1], VASSILIS KOMIS[2],
ANGELIQUE DIMITRACOPOULOU[1], STAVROULA PRANTSOUDI[1]

[1]*Department of Pre-school Education*
and Educational Design
University of the Aegean
Greece
gfesakis@aegean.gr
psemdt16023@aegean.gr
adimitr@aegean.gr

[2]*Department of Educational Sciences*
and Early Childhood Education
University of Patras
Greece
komis@upatras.gr

Abstract

Over the past decade, the assessment of the general educational value of computer programming and computational thinking has been constantly increasing and, as a result, they are introduced to increasingly younger ages. In parallel, educational programming environments are significantly progressing, providing a variety of options for different ages. This paper presents an overview of the modern learning programming environments for primary education and proposes a classification system with categories corresponding to the technological and educational dimensions of the area. The paper aims to support teachers in learning design for the interdisciplinary approach of programming and the development of computational thinking.

KEY WORDS

Educational programming languages, taxonomy, learning design, computational thinking, coding for learning

Résumé

Au cours des dix dernières années, l'évaluation de la valeur éducative de la programmation informatique et de la pensée informatique s'est constamment accrue et, par conséquent, elles ont été introduites à des âges de plus en plus jeunes. Parallèlement, les environnements de programmation éducatifs progressent considérablement, offrant une variété d'options pour les différents âges. Cet article présente un aperçu des environnements d'apprentissage de programmation pour l'enseignement primaire et propose un système de classification avec des catégories correspondant aux dimensions technologiques et éducatives. L'article vise à soutenir les enseignants dans la conception de l'apprentissage pour une approche interdisciplinaire de la programmation et le développement de la pensée informatique.

Mots-Clés

Langages de programmation éducationnelles, taxonomie, ingénierie d'apprentissage, pensée informatique, codage pour apprendre

Introduction

In recent years, computer programming has been at the forefront of the interest to the educational community since it is considered as an activity of great educational value, because both of utilitarian and pedagogical reasons. The utilitarian reasons are related to the predictions concerning the gap between the available jobs related to the Science Technology Engineering and Mathematics (STEM) fields and the number of students who choose to study in these fields internationally (Langdon et al., 2011 in Portelance, Strawhacker, & Bers, 2016; Smith, 2016). Analyst data support the view that economy, in order to innovate and flourish, will need well-trained programmers who will also possess interdisciplinary skills. Furthermore, programming is considered as a key skill for the approaching and understanding of Informatics, Science, and Technology in general. Therefore, by integrating computer programming into general education, the number of students who will possibly choose future studies in STEM cognitive subjects is expected to rise.

In parallel, pedagogical reasons for the integration of programming into general education are projected. Papert (1980) appears as a pioneer supporter of the general educational value of computer programming, and according to him programming can develop a series of higher forms of thinking such as problem solving, analytical and creative thinking. At the same time, DiSessa (DiSessa & Abelson, 1986; DiSessa, 2000) considers programming environments as reconstructive media (Reconstructive Computational Media) or as enhanced or enriched written languages, which allow their

users to organize their thinking with clarity. While computationally tackling a problem, programmers try to "teach" a solution of the problem to the "learner" of Informatics by expressing, observing and clarifying their thinking, and receiving feedback from the execution of the solution, thus making not only the development of a solution, but also the cultivation of metacognitive skills easier for them. Based upon these views, Guzdial & Solloway (2002) also present programming as a modern form of literacy and so do the creators of the Scratch programming environment (Resnick et al., 2009).

A critical decision related to the successful integration of computer programming into education concerns the selection of the proper language or programming environment to be used. The problem of choosing the appropriate educational programming environment is a difficult one, considering that the whole set of available languages is multidimensional and multitudinous. In addition, the basis of the problem has been broadened as it now concerns, not only teachers and educational designers, but also parents, with different priorities in each case. A classification of the available languages based on educational criteria may assist in solving the problem of selection. Pre-existing/Previous classifications of programming environments available (Fessakis & Dimitracopoulou, 2006; Kelleher & Pausch, 2005) need to be updated to include the more recently available options and conceptually cover modern educational approaches. In this paper we propose an up-to-date classification of the educational programming environments, taking into consideration the teacher, the educational designer, and the parent-guardian. We mainly focus on the educational environments and languages that have been built for learning in the context of different scientific fields, and not just for learning programming itself as an end. The following sections present the theoretical background to support the different factors of the programming language selection, followed by the proposed classification, and finally a discussion of the classification and the emerging research directions.

Theoretical background

The problem of programming language selection

The field of programming languages is multidimensional, including a big number of programming languages, as can be seen by examining relevant collections (History of programming languages, 2019; Levenez, 2017; Timeline of programming languages, 2019). In addition, modern software systems are often developed using a combination of programming languages. As a result, programming environments which are independent of a specific language or which can support the programming process for different languages have emerged. Therefore, the programming environment is something different from the programming language (Guo, 2017). For example, the Java language can be used with a simple text editor, or the NetBeans environment, or the Eclipse

IDE, and so on. Similarly, the syntax of the Logo language can be the basis for different programming environments, such as pencilcode.net or turtleacademy.com.

Considering that learning a programming language requires the investment of a considerable amount of time, selecting the programming language and/or the programming environment to be used has proved to be a crucial issue. The selection of the programming environment for introducing primary school students into computer programming constitutes the problem approached in the present paper. If the purpose is the development of professional programming skills, the problem is very different from the case of primary education, where programming is mainly used as an epistemological tool in the context of computational thinking (Fessakis, Komis, Mavroudi, & Prantsoudi, 2018). In the case of primary education, the classification of educational programming environments seems to be more useful than the classification of languages using technological criteria.

Given that computer programming in primary education is considered as a general educational asset, the criteria of selecting a programming environment and corresponding educational tasks should mainly focus on pedagogical and didactic dimensions. The developmental suitability of the environment and the support it provides to the students' thinking and ability of creative problem solving, is given higher priority. Also, the potential of creative expression through digital art forms is equally important to mathematical problems. Whereas the programming model is selected based on the learning ease and the problem area that makes them accessible to the students.

Therefore, from an educational point of view, it would be interesting to classify the available programming environments for beginners, which could provide information on issues such as: which problem fields could be used to produce learning activities, which are the cognitive requirements of the environment, the age range for which the environment is appropriate, how oriented to the computing engine or the problem area the system of the adopted program representation is, which languages and which program models are supported, if implementation of interdisciplinary activities is possible, consistency with modern teaching and learning concepts, etc. Based on these dimensions, we define the main axes of the educational programming environment area described in the next section.

Axes of the educational programming environment area

Considering the technological categories of the programming languages, the pedagogical features which govern the problem of selection and the need to group existing environments by rules of relevance and similarity to facilitate the selection of the proper educational programming environment, five basic axes of analysis of the area, similar to previous classifications (Fessakis & Dimitracopoulou, 2006) are proposed:

A. *Axis of grading of the computing system 'abstraction'*: The position of an environment on this axis gives us an idea of how close the generated programs are in the problem area compared to the computing machine area. The lower the level of abstraction of the computing system of an environment, the more the produced programs use terms that depend on the machine architecture (e.g. register, adder, memory slot, etc.). In contrast, in high-level abstraction environments the produced programs use terms of the problem area (e.g. velocity variable, etc.). The abstraction level of an educational environment determines the possibility of using it in general learning activities, in the context of various subjects, or whether it will be mainly used for programming and computer architecture training.
B. *Axis of developmental-age suitability*: This certain axis provides information about the age range that is appropriate for the use of each environment. The age distribution of the programming environments can, to some extent, be dictated by the general skills they require, but its empirical validation is more often an open problem.
C. *Axis of supported programming paradigm/model*: Supporting multiple programming models is common in modern environments and allows freedom of expression for the programmer. The programming model is important as it greatly determines the ease of learning and the problem area that can be used in designing learning activities.
D. *Axis of supported syntax programming languages*: Most programming environments adopt and support syntax of programs according to the rules of at least one programming language. Dispute over which languages are more appropriate does not seem to be easily reconciled, therefore the support of multiple alternative syntaxes constitutes an interesting feature.
E. *Axis of abstraction approach to the programming process*: In this axis we place educational environments based on the abstract scheme they dictate to the programmer for the programming process and the program execution. For instance, in most procedural environments the programmer, when coding, looks like he is preparing a series of "commands" for the computer-executor. In contrast, in logic programming, the programmer defines a set of events and a set of rules, whereas the "execution" of the program begins by submitting a query concerning the consistency of a sentence compared to the knowledge base-program. In object-oriented programming, the programmer defines object classes, with attributes and methods, organized in hierarchies, while the program is executed when objects from different classes begin to interact with each other and the user. A popular metaphor which helps to add meaning in programming is that of a theatrical scene. In this case, the programmer looks like a scriptwriter-

director who defines the objects-characters that play a role in the context of a play. The execution of the program can be compared to placing objects on a scene where they can interact, based on their programmed behaviors, with each other and with the program user. The axis of the abstraction scheme of the programming process is educationally significant because it determines the addition of meaning to the programming process, and therefore greatly circumscribes the complexity of the problems which can be tackled, the developmental suitability, and the attractiveness of a programming environment.

In this context, the purpose of the present paper is to propose a classification system of programming environments. The proposed classification aims to ease educational designers to select, based on educational criteria, the most suitable environment, depending on the educational situation (educational design) and also aid researchers studying the didactic value of programming in primary education.

In the following, initially the proposed categories of educational environments for computer programming are presented together with typical examples. Then, the categories regarding the axes mentioned above are discussed. Finally, the presentation of the categories is summarized, relevant concluding remarks are presented, and research directions are proposed.

Educational programming environments

Logo family programming environments

Environments in this category are based on the Logo language and the eternal ideas of Papert (Papert, 1980). The dynamics of Logo concepts is based on their ability to adapt so as to be appropriate for different ages; additionally, they allow for the interdisciplinary approach of concepts and problems in order to be applicable in the teaching of various subjects. The Logo language and Paperts' turtles are still used and evolving while inspiring the creation of other systems that are presented below, organized in subcategories.

Roamers

Systems in this category are often referred to as *floor turtles* (Figure 1) (e.g. of the companies https://www.terrapinLogo.com and http://www.valiant-technology.com). These are tangible implementations of the Logo language "turtle", which are able to accept simple commands of movement and orientation (one step forward, one step backward, turn a few degrees to the right or left). Frequently roamers can leave a trace while moving, using markers. Roamers, in practice, take various forms such as bees, cars, etc.

FIGURE 1

Educational Robots - Bee-Bot/Pro-Bot/Blue-Bot roamers of TerrapinLogo

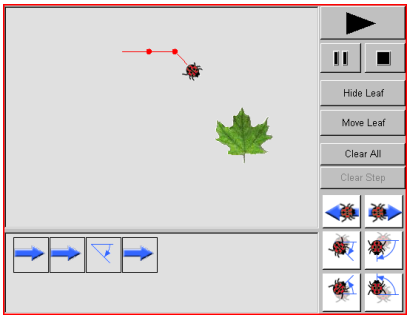
Learning activities usually implemented in such systems, include: a) exit of the roamer from a maze which can be painted on a mat and/or created by obstacles (e.g. styrofoam), b) point-to-point movement (path execution), c) drawing shapes, etc.

These systems are preparation environments for introducing students of the first grades of primary education in more complex and realistic programming environments, of the same family at least. In addition to the ease of use and administration, roamers are popular because of their compatibility to the curriculum and in particular to the concepts of orientation (front, back, right, left), distance (far, near), and numeracy, due to the distinct nature of movement which is based on fixed-spaced steps and angles (e.g. square or 45°). The lack of a requirement of writing and reading knowledge, computer use, and complex constructions such as robotics kits, makes the roamers ideal for the young ages of 3-6 years old. The purposefulness of using roamers in kindergartens is also supported by relevant studies (Pekarova, 2008; Komis & Misirli 2016; Komis, Romero & Misirli, 2017).

Software roamers

Software roamers are created as simulations of physical roamers with a program on a computer. The roamer is replaced by a software entity (an agent) illustrated in various forms, such as a turtle, a vehicle, and so on. These environments allow for the children's gradual distancing from floor turtles and their approach to the typical Logo. The basic training activities remain the same; however, the implementation of computational problems, such as those of a typical programming lesson, is also feasible.

FIGURE 2



User interface of Ladybug Leaf

FIGURE 3

User interface of LCSI Microworlds JR software

The main advantage of direct providing of visual feedback remains. In some cases, environments in this category offer useful tools, such as the virtual protractor, which facilitate programming and linking to other subjects like mathematics. Software roamers cover the gap between children who have learnt reading and writing and children who do not possess or are in the process of acquiring these basic skills (ages 5-7).

Typical examples of this category are the MicroWorlds JR (http://www.microworlds.com) (Figure 3) environment of the LCSI company and the NLVM micro-application series named Ladybug (Figure 2), which are freely available in the National Library of Virtual Manipulatives collection of the UTAH State University (http://nlvm.usu.edu/). In the case of Ladybug the commands are represented with virtual tiles and the program is formed as a sequence of tiles (tile programming). The user can modify the program by adding or removing commands and execute all of it, or up to a point. These environments have been experimentally studied, with positive results in terms of their attractiveness and learning value (Fessakis, Gouli, & Mavroudi, 2013).

Logo programming environments

Indicative of the impact and timelessness of the Logo language is the fact that new programming environments are being developed for it, in all operating systems, even today. Several of these are freely accessible via the Internet and are accompanied by extensive documentation and lively user communities which facilitate learning. A key characteristic that distinguishes them from the systems of the previous category is their textual orientation in programming. They use a text editor to compose programs and an area-window to view the results of the program execution (usually tortoise drawings).

Older systems in this category include (a) free-of-charge simple implementations, such as the KTurtle developed under the KDE Edutainment project (http://edu.kde.

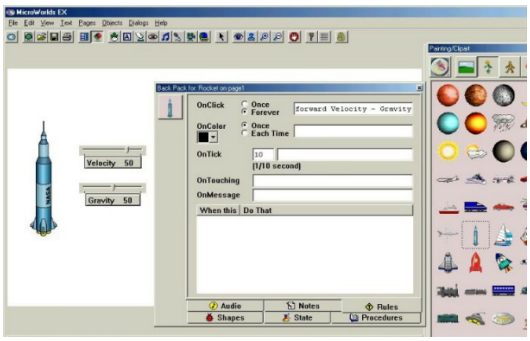
org/kturtle/) or the MSWLogo (http://www.softronix.com) and the FMS Logo (http://fmsLogo.sourceforge.net/), and b) commercially available versions such as Terrapin Logo (https://www.terrapinLogo.com).

Latest implementations are environments such as the Pencilcode (https://pencilcode.net), the Turtle Academy (http://www.turtleacademy.com/), the Turtle Art combining Logo with visual programming (http://turtleart.org/), the MaLT+ (http://en.etl.ppp.uoa.gr), the NetLogo (http://ccl.northwestern.edu/netLogo/) with multiple turtles and a large collection of examples of scientific simulations and the StarLogo TNG/NOVA (http://education.mit.edu/) with emphasis on complex dynamic systems. The contexts of this category can be used to develop algorithmic thinking, CT, and introduction to advanced programming concepts, such as *variable, subprogram, parameter passing, variable range, structured programming, event-driven programming, concurrent programming*.

Expanded Logo environments - Microworld development environments

These are complex microworld development environments combining the Logo ideas with those of tile-based software development. Example of this category is the LCSI Microworlds environment (http://www.microworlds.com/solutions/mwex.html) (Figure 4).

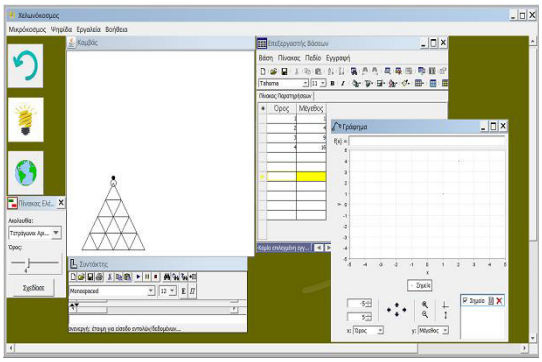
FIGURE 4



User interface of the LCSI Microworlds EX *software*

This environment provides enriched multimedia editing capabilities, a complex graphical interface, and features to support the constructivist approach to programming as a learning tool. Version Logo Microworlds EX was one of the first educational programming environments to work with LEGO Mindstorms educational robotics environment. The ability to use Logo in educational robotics makes it even more attractive as a pedagogic infrastructure investment.

FIGURE 5



Example of microworld with Logo in e-slate

Another notable environment in this category is **E-slate** (Figure 5). According to its designers, *E-Slate is an exploratory learning environment. It provides a workbench for the creation of highly dynamic software with rich functionality, by non-programmers. Educational activity ideas can be turned into software with minimal authoring effort in the form of interactive Microworlds that contain specially designed educational components. Microworlds software can be easily constructed by plugging components in various configurations. The behavior of both components and Microworlds can be programmed into a Logo-based scripting language* (http://e-slate.cti.gr/).

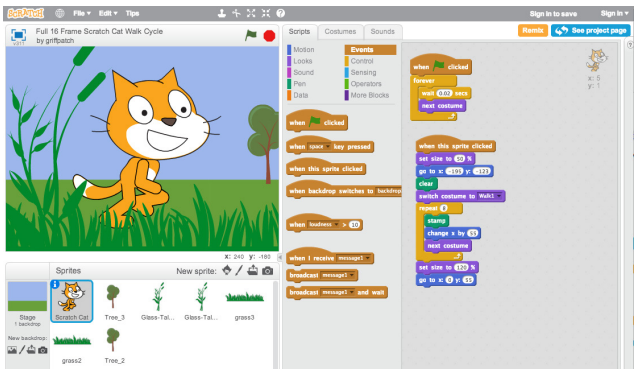
Visual programming environments for kids

Environments in this category are mainly featured by the fact that they use visual programming techniques. The programming language elements are visually represented by blocks which are assembled into more complex syntactic structures and eventually into programs. The use of keyboard is limited to entering parameters into commands. The representation of syntactic limitations of the language in the structure of the building blocks prevents the occurrence of syntactic errors because assembling can be feasible only in such a way that the produced command is syntactically correct. The building blocks are selected from organized pallets and so users do not need to memorize dozens of commands before they focus on programming. Languages are interpreted, and commands can be tested even when the program is incomplete. The model of programming is driven by events, simultaneous procedural programming, while the metaphor of the machine for the programmer is the object control in a theatrical scene. Another important feature of these systems is that their learning is not dependent on knowledge of the English language, thus facilitating even prereaders in their use.

Scratch

Scratch (Resnick et al., 2009) was developed by the Lifelong Kindergarten Learning Group at MIT in 2007. It is a programming environment in which users create programs based on the model of the theatrical scene, using a visual programming language. Programmers have a scene (central screen of the application) at their disposal (Figure 6) in which they create objects (actors and scenery) by choosing from a collection or drawing their own.

FIGURE 6



User interface of the Scratch 2 software

Objects on the scene can interact with each other and with the user based on a predefined by the programmer behavior. The behavior of the objects is defined by dragging building blocks that represent actions-commands which refer to an object. These building blocks comprise the Scratch programming language (Maloney et al., 2008; 2010). Scratch as a visual programming language facilitates programming, experimenting and tinkering. Scratch enables the creation of electronic games, cartoons, interactive stories, and more. It allows users to share their creations on the Internet, in a lively community located at the web address: http://scratch.mit.edu. The design of Scratch intentionally favors novice programmers. In addition to producing applications, as an educational environment it aims at developing basic skills, such as *creative thinking, clear communication, systematic analysis, efficient collaboration, iterative-progressive design, and lifelong learning skills*. Newer versions of Scratch are online and do not need installation. A variation of the Scratch language is Snap! (Build Your Own Blocks) (http://snap.berkeley.edu/) which allows for, among other things, the definition of subprograms, a feature that is not available in Scratch.

ScratchJr

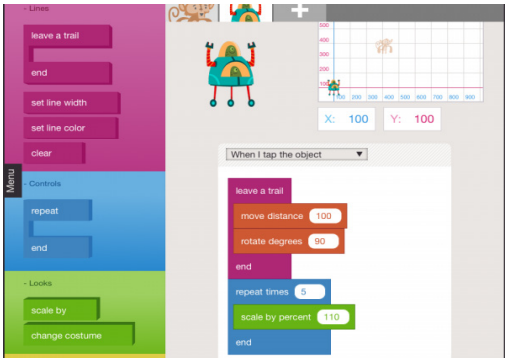
ScratchJr (http://www.scratchjr.org/) (Portelance, Strawhacker, & Bers, 2015) is a variation of the Scratch programming environment for younger children. ScratchJr

was developed by MIT in collaboration with the DevTech Research Group at Tufts University, after a major redesign to be adapted to the needs and abilities of children aged 5-7 (Figure 10). ScratchJr (Figure 7) is an introductory programming language which allows even pre-school children to develop games and interactive stories. It implements the logic of Scratch, where programming is done by sequencing and nesting tiles which represent the statements and commands of the language, and also supports the metaphor of the scene, whereas children can draw characters and scenes and record or compose sounds. ScratchJr is distributed as an Android and iOS application and targets at the generation of children growing up with tablets. ScratchJr paves the way for research on the Didactics of Programming, Computational Thinking, and the effects of its use in preschool age.

FIGURE 7

User interface of ScratchJr

FIGURE 8



User interface of Hopscotch

Hopscotch

Hopscotch (https://www.gethopscotch.com/) is an educational and entertaining programming environment for children, with great potential and a conceptual structure similar to Scratch. As in Scratch, computer programming is approached as a means of creative expression for everyone. With Hopscotch it is possible to explore and approach key concepts of Informatics, such as: *abstraction, variables, logical expressions, conditional branches, repetitions* etc. in a playful environment. On the website there is a plethora of examples and supportive material. Hopscotch (Figure 8) is more relevant to Scratch than ScratchJr, but until the advent of Jr it was perhaps the only equivalent in iOS environment.

Blockly

Scratch made programming with blocks so popular that Google has developed a library for creating such languages at will. Blockly (https://developers.google.com/blockly/) is a client-side JavaScript library for creating visual block programming languages and editors. It is a project of Google and is open-source under the Apache 2.0 License. It typically runs in a web browser, and visually resembles Scratch (Fraser 2015; Pasternak, Fenichel, & Marshall, 2017). In essence, using this library one can create Scratch-like languages and environments with great ease. Examples of educational applications with Blockly can be explored at: https://blockly-games.appspot.com.

Logic and object-oriented programming environments

In this category we quote educational programming environments which adopt a programming model different from the usual procedural one. For example, there is Toontalk (http://www.toontalk.com/) (Morgado, 2005; Morgado & Kahn, 2008; Morgado, Cruz, & Kahn, 2003), proposing a special approach to programming for preschool children in which extensive imaginative metaphors for programming concepts framed in a virtual cartoon world are used. A similar, interesting, different programming approach with visual rules for young children is provided by the Viscuit environment (Harada & Potter, 2003). In parallel, Agentsheets (Repenning & Ioannidou, 2004) and Stagecast (Seals et al., 2002) systems support a more common programming model by combining agents, visual programming and logical rules such as those of the Prolog language. It is reasonable for these systems to be considered as appropriate for preparing introduction to logic programming with Prolog, or languages such as Racket (http://racket-lang.org/) which combines Scheme and Lisp.

Languages which fully adopt the model of object-oriented computer programming constitute a special subcategory. Predominantly based on the Smalltalk object-oriented language, the Squeak language was initially developed (Guzdial & Rose, 2001; Ingalls et al., 1997) (http://squeak.org/) and, based on this, the educational e-toys environment (http://www.squeakland.org/), making it more accessible to younger children. Programming

in Squeak e-toys (Morge, Narayan, & Tagliarini, 2010) (http://etoysillinois.org/) is very different and it is worth experimentally investigating its effect on general education in the light of computational thinking development. It is worth noting that Scratch was originally developed in the Squeak language, just as e-toys. Also noteworthy is the fact that e-toys is a main educational programming environment included in the student computer of the OLPC project (http://one.laptop.org/) along with TurtleArt, Squeak and Scratch.

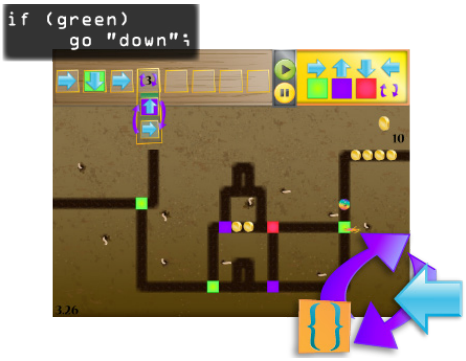
Commercial programming learning environments for entertainment purposes

The spread of children's education in programming and the attribution of entertainment character to relevant activities, a series of commercially available environments for learning programming has been produced. These environments present a selective collection of design choices of the respective research environments without proposing certain innovations. Commercially available environments raise issues of social inequalities on the access to learning which, however, are not essential since there are also free, high-quality environments available. Furthermore, environments in this category show a tendency to commercialize children's entertainment and education through the Internet and ICT. Some of the most well-known systems are indicatively presented here for the purpose of completeness of the categorization system. Their range of topics concerns a wide variety of activities, from familiarization with a specific programming language, problem solving with Logo-like code, to game creation (e.g. http://stencyl.com/).

Kodable: Kodable (http://www.kodable.com/) is a commercial programming learning environment with tiles that is accompanied by an integrated teaching administration system. The parents' arguments are related to the future ability of finding a job. The programming language looks more like that of LadyBug (described below), rather than that of Scratch. Students are asked to program the exit from a labyrinth of an agent which accepts a limited repertoire of simple commands (Figure 9). The use of Kodable is possible even before someone learns writing or arithmetic. It is accompanied by a curriculum in which, in addition to general programming skills, objectives from other curricula, such as Mathematics, are also cultivated. It runs on iOS.

Tynker: Tynker (https://www.tynker.com/) is a commercially available integrated teaching support system on computer programming through the Internet. The system includes a programming environment with Scratch-like tiles, which is accompanied by detailed learning activities organized in full curricula. Its use facilitates parents, teachers and principals in managing courses and monitoring results. By providing detailed lesson plans, this certain environment promises applicability even by teachers with little programming knowledge.

FIGURE 9



User interface of Kodable

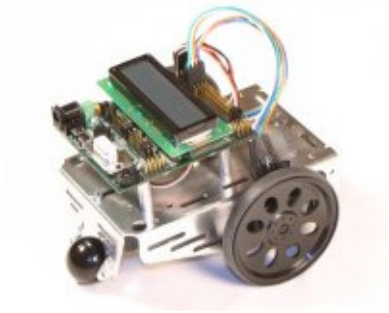
Physical Computing environments

The term Physical Computing refers to the broad field of interactive physical system development, that is electro-mechanical compositions controlled by software and using sensors, motors, switches, and other elements to respond to the analogue world. Programming and Computational Thinking are basic components of physical programming and so are electronics and engineering. In general, physical computing is a basic platform for the integrated implementation of the STEAM approach and for this reason it has also been at the forefront in recent years. In this section we present subcategories of physical programming environments. It appears that the term physical programming comes to combine a series of similar areas of Informatics applications that pre-existed.

Educational robotics environments

Educational robotics kits include some programmable microcontroller-microprocessor, connected with inputs and outputs to a printed circuit board, which can be programmed using a computer programming environment. The program is transferred from the computer to the microcontroller and can be run autonomously until a new one is installed. On the main circuit of the microprocessor, devices such as motors, lamps, switches, etc., as well as sensors (of temperature, light, sound, etc.) are connected for the assembly of automated control systems, which usually have moving parts or also comprise whole self-moving ones. The construction of a robot in this way is a rich, interdisciplinary activity involving Physics, Mathematics, Electronics, Mechanics and Informatics problems (Figures 10-12).

FIGURE 10

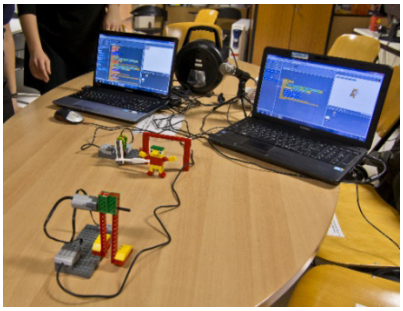


The educational robot IntelliBrain-Bot by RidgeSoft

FIGURE 11

Robot constructed with the LEGO Mindstorms system

FIGURE 12



Constructions with Lego WeDo and Scratch

The cost of the Robotics kits, and physical programming in general, is an inhibiting factor for its massive spread, yet it reaches the students through various promotional policies.

Typical examples in this category include the LEGO Mindstorms (Figure 11) and Wedo (for younger children) kits (Figure 12), the IntelliBrain-Bot by RidgeSoft (IntelliBrain™-Bot) (Figure 10), the microcontroller, sensor, and device kits of Parallax with industrial specifications, the VEX Robotics, the Thymio robot (https://www.thymio.org) for easy introduction without any manufacturing challenges, and the pre-school KIBO (Sullivan, Bers, & Mihm, 2017).

The activities proposed in the context of educational robotics are widely varied, interdisciplinary and they integrate programming into the STEAM approach. They may include, from the problem of maze exit, to the creation of robotic pets playing board games or musical instruments. Some indicative possible projects are described in detail in Ferrari, Ferrari & Hempel (2001), while on the internet there is a plethora of available

resources, varying from ideas for constructions up to complete curricula. There are also communities of enthusiastic amateurs who can keep the interest unabated, while the combination of robotics with Rube Goldberg machines (https://www.rubegoldberg.com/) can provide an entertaining dimension to the range of topics.

Electronics, automatic control and signal processing environments

The category of physical programming includes environments emphasizing on hardware programming. The lowest level of abstraction concerns programming at the level of circuits. At the next level, there are the control systems (of data collection, signal processing and device driving) through a PC or their simulations. At the top level, there is the assembly of systems controlled by stand-alone programmable microprocessors. These systems are also referred to as embedded or internet of things, and are complex systems that concern the upper grades of primary school. In what follows, we describe the subcategories and provide examples of programming environments.

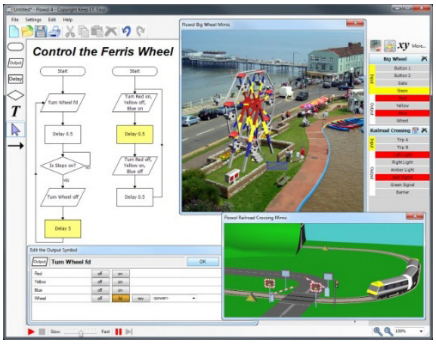
Circuit simulators

In circuit simulators the implementation of algorithms occurs on a physical level, with hardware, and requires the understanding of computer architecture, electronics concepts of automatic control, information representation and signal processing. Indicative systems of this category are LTspice (http://www.linear.com/solutions/ltspice), TINA (https://www.tina.com/), and MultiMedia Logic Simulator (https://sourceforge.net/projects/multimedialogic/), along with the Technology package by Yenka (https://www.yenka.com/technology/). Multimedia Logic Simulator is an older educational package for experimentation and exploratory learning which unfortunately is not supported enough any longer, but it is mentioned here because its design aims to project digital logic as a programming language and because it has been accompanied by remarkable books and relevant educational material, such as the work of Maxfield (1998; 2009). Perhaps in the future, this direction of programming at a physical level will recover the interest of the educational community, combined with modern platforms such as Arduino.

Automatic control systems and their simulators

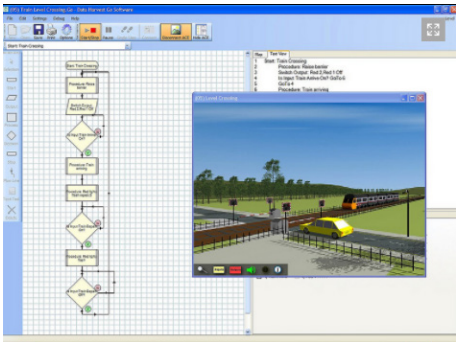
Another series of environments concerns systems that emphasize on the concept of automatic control. This concept is certainly included in educational robotics systems, however, systems in this category usually avoid the complex construction problems of robots. In automated control system activities students focus on the logic-programming of a control system, which is a generalized system consisting of a set of inputs and outputs.

FIGURE 13



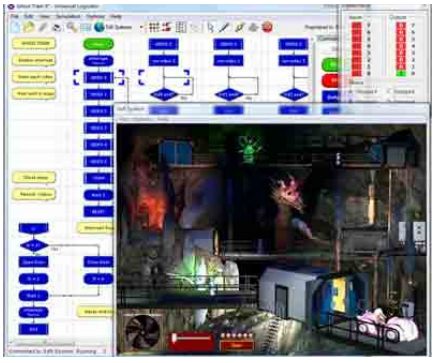
User interface of the Flowol software by Robot Mesh

FIGURE 14



Go Control Software by Data Harvest Group

FIGURE 15



User interface of the Logicator software by Economatics Education Ltd

Students do not program a computer but they specify a control system of another system. A widespread representation system in environments of this category is reported to be the flowchart, while the state diagrams for finite state automata and the logic system flowcharts are also used. The logic representations of the systems to be controlled are usually called 'mimics' and resemble microworlds. For example, instead of students controlling a real greenhouse in order to achieve ideal plant growth conditions, they can control a virtual representation of it on the computer. Some examples of environments in this category include Flowol by Robot Mesh (Figure 13), Go Control Software by the Data Harvest Group (Figure 14), and Logicator software by Economatics Education Ltd (Figure 15).

Miscellaneous unplugged applications and environments

This category includes learning scenarios for programming that do not need a computer to be implemented (unplugged), such as those from: the collection of the Csunplugged project (http://csunplugged.org/) and the Bebras organization (http://bebras.org/). We also include activities from other institutions' initiatives, such as Code.org (http://code.org), the Code-to-Learn Foundation (http://codetolearn.org), and CoderDojo (http://coderdojo.com). This category also includes a series of board games, through which children can be introduced to programming or practice their knowledge of it (Crawley, 2014). In the category various titles (some are games) are also included, such as: *Lightbot Jr 4+ Coding Puzzles (1-3), Move the Turtle. Programming for kids (1-6)*. The grades for which they are considered to be appropriate are reported in the brackets.

Summary of the educational programming environments categories

In this section, we briefly summarize the classification of educational programming environments for primary education and outline the proposed categories in the pedagogical axes described in 'Theoretical background'.

In relation to the 2006 classification (Fessakis & Dimitrakopoulou, 2006), it is now apparent that from the era of the dominance of Logo and its adaptations there has been a transition to the era of a plethora of different commercial and free options for introducing children, even from pre-school age, to programming and computational thinking. The options range from games, to robotics kits and integrated teaching systems. The effect of Scratch, and visual programming in general, on the design of most systems is evident. The prevention of syntax errors and the visualization of the code, which mainly contribute to visual programming languages, had a great impact on the field. Moreover, modern environments support multimedia, different programming models and new forms of learning activities. The category of visual programming environments

combined with the physical programming category with educational robotics kits and microprocessors is today at the forefront of the educational interest as it is a platform for the STEAM approach (Blikstein 2013; Przybylla & Romeike, 2014) and CT development. Another major development is the category of comprehension activities for programming concepts without the use of computers (unplugged). Comparing the above, it is clear that the landscape is much different since the previous classification (Fessakis & Dimitrakopoulou, 2006). The effects on research and education practice are significant. The pedagogical axes categories presentation follows.

A. Axis of grading of the computing system 'abstraction'

Educational programming environments diversify depending on how much they require the programmer to think in terms of computing machine architecture or in terms of the problem area. In cases we focus on using computer programming for problem-solving activities, we are interested in systems with a high level of abstraction from the computing machine. On the contrary, there are cases in which we wish to focus on details of the implementation of programming, so we are interested in transparency towards the machine. In the extreme case of low abstraction, which is of minimum concern in primary education, we identify the programming environments in symbolic language, the physical programming environments, and the robotics environments, whereas at the high levels of abstraction we identify the visual programming environments, the Logo environments, and the unplugged computing environments.

B. Axis of developmental-age suitability

In Table 1 the educational programming environments and the respective ages proposed for their implementation are presented. Table 1 also presents the arrangement of the categories in the axis (B). Age correspondence is indicative and based on manufacturers' suggestions and authors' estimates.

Other dimensions that are of interest in selecting an educational programming environment are the programming model it supports, the language whose syntax it uses, and the metaphor it uses for programming as a whole process. The presentations of the corresponding axes (C-E) (Section 2.2) are analyzed next.

C. Axis of supported programming paradigms/models

Supported models of educational programming environments include procedural, visual, event-driven, object-oriented, parallel, concurrent, distributed, integrated, logic, and programming with Artificial Intelligence techniques, etc. The programming model is important as it determines the expressive power of the environment and the problem area that can be used to design activities. Therefore, familiarizing students with multiple models is an advantage (Stephenson et al., 2005) because it provides them with alternative ways of computational thinking. In addition, each model has different

requirements from the student in its conquest and implementation, and research in Didactics is needed for the comparison of alternative models. The procedural model has been studied disproportionately more than the rest, especially at the primary education level.

TABLE 1

Developmental suitability of educational programming environments

	Age intervals in years			
Educational Learning Environment	**4-6**	**6-8**	**8-12**	**12-15**
Logo FAMILY				
ROAMERS	✓			
SIMULATED ROAMERS		✓		
Logo ENVIRONMENTS			✓	✓
EXP. Logo - MICROWORLDS			✓	✓
GEN/ZED TURTLEWORLDS				✓
VISUAL PROGRAMMING	✓	✓	✓	✓
PHYSICAL PROGRAMMING				
ED. ROBOTICS	✓	✓	✓	✓
ELECTRONICS/A. CONTROL		✓	✓	✓
MICROCONTROLLERS				✓
LOGIC & OBJ.-OR. PROGRAMMING			✓	✓
PSEUDOCODE				✓
UNPLUGGED	✓	✓	✓	✓

D. Axis of supported syntax programming languages

Several times the programming language that one is asked to use is given and is not left as an option to the teacher. Language selection is often considered essential in order, for example, to utilize the time a student will spend on a language that is considered competitive in terms of market demands, or can be used in other courses as well. Of the languages used in the environments, the more widely known ones are Logo, Basic, Pascal, Java, and Javascript which are considered older, while more languages, such as Python, Ruby and Blockly-like graphical languages have been added. Programming

environments which simultaneously support more than one language are of particular interest.

E. Axis of abstractional approach of the programming process

Each environment assumes and displays an abstractional approach for the programming process. This approach affects students' perception of the meaning of programming. From this perspective, roamers show their programming as a process during which *"I make the roamer do something"*. Then, the student interacting with a Logo turtle is asked to *make an abstract software entity do something*. The next model introduces certain turtleworlds that support the use of multiple turtles (or miniature-robots), thus programming appears as a process during which *"I make entities interact in order to do something."* In the case of educational robotics, students follow the previous schema, not with software entities, but with devices constructed by themselves, so in this case programming can be described as an activity during which *"I make devices that interact to do something"*. In automated control systems the student *"controls a system"*. The theatrical scene metaphor is a very convenient abstractional schema for introducing children to simultaneous, event-driven programming. It is the Scratch model that has pinpointed programming for children since first grade of primary school.

Conclusion-discussion

Computer programming is considered a learning and developmentally beneficial activity for children (Papert, 1980; DiSessa 2000). Through engagement with programming, children can develop higher forms of thinking such as problem-solving ability, creative thinking and metacognitive skills. Selecting the appropriate educational programming environment is a critical decision. The overview of educational programming environments ascertains a variety of available systems which can cover the full range of student primary school ages, concern the most common programming models, and utilize the most known programming languages. Each environment adopts a different schema for the process and the purpose of programming, the simplest being that of roamers and the most complex that of educational robotics. For a long time, Logo dominates as a stand-alone programming language and as a control environment for educational robotic arrangements in children's programming. The increasing interest in the development of CT, along with the development of modern, specially designed environments for children's programming, have changed the landscape and require adaptation of the research agenda and educational practice. The proposed overview and classification of educational programming environments is expected to facilitate the educators' teaching design and orientate the related educational research. In particular, the overview and classification can highlight some of the most significant developments in the field of educational programming environments such as the following:

- Programming environments are now multimedia environments, using a theatrical scene and "actors" as a basic model-metaphor for the programming. They are not just about moving the turtle and controlling its trail. Modern environments with the use of visual programming languages make coding easy, while they require minimum typing and memorizing of commands and syntactic rules. Particularly, syntactical errors are completely avoided, thus making a series of research topics of didactics outdated (Pea 1986; Spohrer & Soloway 1986; Stephenson et al., 2005).
- The types of applications-tasks that pupils are asked to concern with are very diverse and rich. After the turtle graphics and mathematical/geometrical problems, young programmers are now mainly aiming at creative expression and entertainment by creating artifacts such as interactive cards/posters, stories, animations and games.
- Today's programming environments can be used even before familiarization with writing and reading words and numbers. Thus, the question about the indicative teaching order and the manner of activity interactions arises.
- Children's programming environments and supportive material are now available not only on desktop computers but also on portable devices (tablets/iPads), rendering programming accessible outside school too and parent involvement more feasible.
- There is an increasing tendency to commercialize programming learning, with several environments circulating as commercial products. This point makes the role of public education more essential, aiming at the elimination of potential inequalities.
- Finally, a series of educational robotics product collections in the form of advanced games is available for young children, giving them more opportunities of engaging in more integrated, specific and authentic activities that will familiarize boys and girls with the basic concepts and methods of digital technology.
- Physical programming is expected to spread more over the next few years, along with the development of the STEAM approach and the interdisciplinary approach in general.
- Computational Thinking provides a conceptual framework for the integration of programming into the various subject areas. The use of scientific computing environments and computing modeling in education will be further extended.
- Artificial Intelligence and dynamic data analysis will become the new scope of computer programming in primary education. They are elements of the modern world with great economic and social consequences and investment in related knowledge has been rendered essential.

The combination of the aforementioned ascertains the fact that the available technology has surpassed in pace the corresponding educational research, whose main body concerns the Logo language in primary education. New research directions, in relation to effectiveness of programming environments, their long-term effects, transition to adult programming environments, curricula, learning approaches, didactics of computational

thinking, types of learning activities, teachers' role and preparation, etc., can be realized and contribute to the optimal utilization of programming teaching at this educational level. Considering the above analysis, we can describe new research directions for the Didactics of computer programming and, in general, Computational Thinking in primary education. Particularly, it is useful, under the Didactics of Informatics perspective, topics such as the following to be studied:

- Comparison of the impact of different programming models on familiarizing students with programming (cognitive difficulties, proposed interventions, performance, etc.)
- Comparison of the effect of the abstractional approach of the programming process on familiarizing students with programming.
- Comparison of the effect of the type of task-result (e.g. mathematical algorithms, game, interactive story) on familiarizing students with programming.
- Emotional factors in programming learning. For instance: which emotions are associated to each environment.
- Interdisciplinary approach of computer programming and its role in the STEAM approach by studying and improving the comprehension of concepts involved in related cognitive fields, such as Mathematics and Science.
- Study of the transition from visual programming to text programming and from educational to professional one.
- Study of the interaction of programming learning with other general skills such as problem-solving skills, spatial thinking, language and creativity.
- Emergence of the possibilities and boundaries of different environments as means of constructing digital artifacts of expression, thought and mental experimentation by students.
- Study of ways to remove stereotypes and other barriers for access to programming by both genders.

Several of the research topics proposed in the present paper remain open since the previous classification (Fessakis & Dimitrakopoulou, 2006). The systematic research on the Didactics of programming and Computational Thinking will contribute to the better utilization of the available wealth of programming environments in the school context.

References

Blikstein, P. (2013). Gears of our childhood: Constructionist toolkits, robotics, and physical computing, past and future. In *Proceedings of the 12th International Conference on Interaction Design and Children* (pp. 173-182). New York, USA: ACM.

Crawley, D. (2014). *12 games that teach kids to code – and are even fun, too.* Retrieved from http://venturebeat.com/2014/06/03/12-games-that-teach-kids-to-code/.

DiSessa, A. (2000). *Changing minds. Computers, Learning, and Literacy*. Cambridge, MA, USA: MIT Press.

DiSessa, A., & Abelson, H. (1986). BOXER: A reconstructive computational medium. *Communications of the ACM*, *29*(9), 859-868.

Ferrari, M., Ferrari, G., & Hempel, R. (2001). *Building robots with Lego mindstorms: The ultimate tool for mindstorms maniacs*. Rockland, MA: Syngress.

Fessakis G., & Dimitracopoulou A. (2006). Review of educational environments for programming: Technological and pedagogical dimensions. *THEMES in Education, 7*(3), 279-304.

Fessakis, G., Gouli, E., & Mavroudi, E. (2013). Problem solving by 5-6 years old kindergarten children in a computer programming environment: A case study. *Computers & Education, 63*, 87-97.

Fessakis, G., Komis, V., Mavroudi, E., & Prantsoudi, S. (2018). Exploring the scope and the conceptualization of computational thinking at the K-12 classroom level curriculum. In M. S. Khine (Ed.), *Computational Thinking in the STEM Disciplines: Foundations and Research Highlights*. (pp. 181-212). Switzerland: Springer.

Fraser, N. (2015). Ten things we've learned from Blockly. In *2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)* (pp. 49-50). Atlanta, GA, USA: IEEE Computer Society.

Guo, P. (2017). Building tools to help students learn to program. *Communications of the ACM*, *60*(12), 8-9.

Guzdial, M. J., & Rose, K. M. (2001). *Squeak: Open personal computing and multimedia*. River, NJ, USA: Prentice Hall PTR.

Guzdial, M., & Soloway, E. (2002). Log on education: Teaching the Nintendo generation to program. *Communications of the ACM, 45*(4), 17-21.

Harada, Y., & Potter, R. (2003). Fuzzy rewriting: Soft program semantics for children. In *Human-Centric Computing Languages and Environments, IEEE CS International Symposium on* (pp. 39-46). Auckland, New Zealand: IEEE Computer Society.

History of programming languages. (2019). *In Wikipedia, The Free Encyclopedia*. Retrieved from https://en.wikipedia.org/w/index.php?title=History_of_programming_languages.

Ingalls, D., Kaehler, T., Maloney, J., Wallace, S., & Kay, A. (1997). Back to the future: The story of Squeak, a practical Smalltalk written in itself. *ACM SIGPLAN Notices, 32*(10), 318-326.

Kelleher, C., & Pausch, R. (2005). Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys, 37*(2), 83-137.

Komis, V., & Misirli, A. (2016). The environments of educational robotics in Early Childhood Education: Towards a didactical analysis. *Educational Journal of the University of Patras UNESCO Chair, 3*(2), 238-246.

Komis, V., Romero, M., & Misirli, A. (2017). A scenario-based approach for designing educational robotics activities for co-creative problem solving. In *Advances in Intelligent Systems and Computing* (pp. 158-169). New York: Springer.

Langdon, D., McKittrick, G., Beede, D., Khan, B., & Doms, M. (2011). *STEM: Good jobs now and for the future*. Retrieved from https://goo.gl/rmyDfU.

Levenez, E. (2017). *Computer Language History*. Retrieved from https://www.levenez.com/lang/.

Maloney, J. H., Peppler, K., Kafai, Y., Resnick, M., & Rusk, N. (2008). Programming by choice: Urban youth learning programming with scratch. *ACM SIGCSE Bulletin, 40*(1), 367-371.

Maloney, J., Resnick, M., Rusk, N., Silverman, B., & Eastmond, E. (2010). The scratch programming language and environment. *ACM Transactions on Computing Education*, *10*(4)*, 16*, 1-15.

Maxfield, C. (1998). *Designus Maximus unleashed! (unabridged & unexpurgated): Banned in Alabama!* Boston: Newnes.

Maxfield, C. R. (2009). *Bebop to the Boolean boogie: An unconventional guide to electronics*. Amsterdam: Elsevier, Newnes.

Morgado, L. C. (2005). *Framework for computer programming in preschool and kindergarten*. Phd Thesis. Retrieved from http://www.scribd.com/ doc/24041133/Framework-for-Computer-Programming-in-Preschool-and-Kindergarten.

Morgado, L., & Kahn, K. (2008). Towards a specification of the ToonTalk language. *Journal of Visual Languages & Computing*, *19*(5), 574-597.

Morgado, L., Cruz, M. G. B., & Kahn, K. (2003). Working in ToonTalk with 4-and 5-year olds. In P. Isaías & A. Palma dos Reis (Eds.), *International Association for Development of the Information Society - IADIS International Conference e-Society 2003* (p. 988). Lisbon, Portugal: IADIS.

Morge, S., Narayan, S., & Tagliarini, G. (2010). Squeak etoys modeling and simulation tool: Empowering students and teachers to create, explore, collaborate and interact. In Z. Abas, I. Jung & J. Luca (Eds.), *Proceedings of Global Learn Asia Pacific 2010 - Global Conference on Learning and Technology* (pp. 589-591). Penang, Malaysia: Association for the Advancement of Computing in Education.

Papert, S. (1980) *Mindstorms: Children, computers, and powerful ideas*. New York: Basic Books Inc.

Pasternak, E., Fenichel, R., & Marshall, N.A. (2017). Tips for creating a block language with blockly. In *2017 IEEE Blocks and Beyond Workshop (B&B)* (pp. 21-24). Raleigh, NC, USA: IEEE.

Pea, R. (1986). Language-independent conceptual 'bugs' in novice programming. *Journal of Educational Computing Research, 2*(1), 25-36.

Pekarova, J. (2008). Using a programmable toy at preschool age: Why and how? In *Workshop Proceedings of SIMPAR 2008 Intl. Conf. on Simulation, Modeling and Programming for Autonomous Robots* (pp. 112-121). Venice, Italy: SIMPAR.

Portelance, D. J., Strawhacker, A., & Bers, M. U. (2016). Constructing the ScratchJr programming language in the early childhood classroom. *International Journal of Technology and Design Education, 26*(4), 489-504.

Przybylla, M., & Romeike, R. (2014). Physical computing and its scope – towards a constructionist computer science curriculum with physical computing. *Informatics in Education*, *13*(2), 241-254.

Repenning, A., & Ioannidou, A. (2004). Agent-based end-user development. *Communications of the ACM*, *47*(9), 43-46.

Resnick, M., Maloney, J., Monroy-Hernandez, A., Rusk, N., Eastmond, E., Brennan, K., et al. (2009). Scratch: Programming for all. *Communications of the ACM, 52*(11), 60-67.

Seals, C., Rosson, M. B., Carroll, J. M., Lewis, T., & Colson, L. (2002). Fun learning stage cast creator: An exercise in minimalism and collaboration. In *Proceedings - IEEE 2002 Symposia on Human Centric Computing Languages and Environments, HCC 2002* (pp. 177-186). Toronto, ON, Canada: IEEE.

Smith, M. (2016). *Computer Science for all: Learn about President Obama's bold new initiative to empower a generation of American students with the computer science skills they need to thrive in a digital economy.* Retrieved from https://obamawhitehouse.archives.gov/blog/2016/01/30/computer-science-all.

Spohrer J., & Soloway E. (1986). Novice mistakes: Are the folk wisdoms correct? *Communications of the ACM*, *29*(7), 624-632.

Stephenson, C. et al. (2005). *The new educational imperative: Improving high school Computer Science Education*. Final Report of the CSTA Curriculum Improvement Task Force, ACM. Retrieved from http://csta.acm.org.

Sullivan, A., Bers, M. U., & Mihm, C. (2017). Imagining, playing, & coding with KIBO: Using KIBO robotics to foster computational thinking in young children. In *Proceedings of the International Conference on Computational Thinking Education, 2017*. Wanchai, Hong Kong. Retrieved from https://kinderlabrobotics.com/research-articles/.

Timeline of programming languages. (2019). *In Wikipedia, The Free Encyclopedia*. Retrieved from https://en.wikipedia.org/w/index.php?title=Timeline_of_programming_languages.