Article

Koios: Design, Development, and Evaluation of an Educational Visual Tool for Greek Novice Programmers

Journal of Educational Computing Research
0(0) 1–33

DOI: 10.1177/0735633118781776
journals.sagepub.com/home/jec
SAGE

Ioannis V. Vasilopoulos[1] and Paul van Schaik[1]

Abstract

This article discusses the design and implementation of a new programming tool for Greek novices as a means to improve introductory programing instruction in Greece. We implemented Koios, a new highly interactive and visual programming tool for Greek novices, based on the body of research in the field of psychology of programming. The main contribution of this article is the empirical demonstration of the benefit of this tool in novice programming, compared with two other popular programming tools for Greek novices. The results show that users of Koios performed significantly better than users of the other two programming tools.

Keywords

novice programming, constructivism, cognitive load theory, programming tool, evaluation, assessment

Introduction

Learning to program is an important area of computer literacy included in the majority of curricula worldwide. All levels of programming instruction involve devising programming solutions to problems and implementing them into

[1]Teesside University, Middlesbrough, UK

Corresponding Author:
Ioannis V. Vasilopoulos, Orfanidou 11, TK 15126, Marousi, Athens, Greece.
Email: i.v.vasilopoulos@gmail.com

programs using a programming language (PL) and, ideally, an integrated development environment (IDE; Winslow, 1996). However, learning and teaching programming is notoriously reported in literature as a very difficult task, especially for novice programmers (Robins, Rountree, & Rountree, 2003). To alleviate some of novice programmers' difficulties, the design of educational PLs and IDEs aims to better support novices in program creation than professional PLs and IDEs. Taking into consideration, principles that are identified and informed by research in the psychology of programming and other relevant fields could offer guidelines and rules of thumb for producing educational programming software which could provide a supportive learning environment and better facilitate novices in learning how to program.

However, little research has been conducted in this field in Greece, with few exceptions (Efopoulos, Dagdilelis, Evangelidis, & Satratzemi, 2005; Kordaki, 2010; Vrachnos & Jimoyiannis, 2008). To help Greek novices, complete their first programming course, we designed and implemented a new Greek programming tool called Koios for teaching and learning introductory programming. The first part of this article discusses the design and implementation of Koios, which was substantially inspired and heavily influenced by the theories of constructivism and cognitive load theory (CLT) as well as by reviewing novice programmers' difficulties and previous work in the field. Koios' design and implementation also followed a user-centered approach because developing a software tool that helps and supports its users in efficiently accomplishing tasks is particularly important, especially for complex and challenging tasks, like programming. More specifically, Koios' distinctive combination of characteristics (e.g., textual code automatic generation, error prevention, informative feedback, and constant guidance) supports its users in overcoming common difficulties novices face and enabling them to focus on the task of programming. In this context, Koios' design was inspired by similar software such as BACCII (Calloni, Bagert, & Haiduk, 1997), jGRASP (Hendrix, Cross, & Larry, 2004), ALVIS Live! (Hundhausen & Brown, 2007; Hundhausen, Farley, & Brown, 2009), Scratch (Resnick et al., 2009), Alice (Bishop-Clark, Courte, Evans, & Howard, 2007), JEliot (Ben-Ari et al., 2011), and Ville (Rajala, Laakso, Kaila, & Salakoski, 2008). The second part of the article presents the empirical evaluation of Koios and demonstrates the benefits of this tool on learning outcomes in comparison with two popular programming tools for Greek novice programmers. The study involved assessing the programming performance (end-test scores) of three groups of novice programmers, each using one of the three tools in a practical test at the end of their introductory course.

The rest of the article is organized as follows: The next section presents the design and implementation of Koios, which is followed by the section that reports the empirical study used for the evaluation of Koios and its results, then a further section that discusses the study findings, and the final section that concludes this article.

Koios—A New Greek Visual Programming Tool for Greek Novices

Koios was named after a titan from Greek mythology, who was the titan god of intelligence and an eager learner. This section briefly describes the conceptual background, design, and development of Koios, but more details regarding Koios can be found elsewhere (Vasilopoulos, 2014).

Theoretical Background of Koios

Koios' design and development was theoretically supported by two learning theories, namely constructivism and CLT. The two theories together with the most-often cited difficulties that novices face in introductory programming and the existing body of work in the field formed the conceptual framework that inspired and guided Koios' design and development.

Constructivism is a theory introduced by Piaget in 1967, and its claim point is that each individual constructs new knowledge differently (Ackermann, 2001; Crook & Sutherland, 2017; Kay & Kibble, 2016). According to constructivism, due to the lack of an objective truth to which all individuals can refer for acquiring new knowledge, new knowledge is constructed based on already-acquired knowledge and previous experiences that are actively and repeatedly used in this construction (Sjøberg, 2007). Constructivist views emphasize the active role of the learner in the educational environment and the importance of exploratory learning. Schema, the basic building block of knowledge used in this process, is an internal hierarchical cognitive structure that represents a set of attributes of a concept from the real world, such as information, the vocabulary, the actions, and the experiences related to that specific concept.

New experiences and information combined with previous knowledge dynamically add to (process of assimilation) or modify (process of accommodation) existing schemata. Therefore, the two processes construct and modify learners' knowledge according to their experiences and perception of the world.

CLT was introduced by John Sweller (1988) and postulates that the learning process is based on three types of memory (sensory, working, and long term) and three types of cognitive load (intrinsic, extraneous, and germane), which are imposed on working memory by learning processes according to the learner's level of expertise. However, the latest revision of the theory distinguishes only between intrinsic and extraneous cognitive load; depending on the type of load working-memory resources are allocated to, they are characterized as germane resources (intrinsic load) and extraneous resources (extraneous load; Sweller, Ayres, & Kalyuga, 2011).

Sensory memory receives all sensory stimuli from the outside world and are retained for only a small fragment of time and, if not transferred to working memory, are considered lost. Working—or short term—memory is hypothesized

to be the place where all conscious processing of information occurs and to have a limited capacity. However, Paas and Sweller (2012) argued that these limitations are crucial only when novel biologically secondary information is processed. Biologically secondary information is cultural knowledge that humans were not specifically evolved into acquiring and originates from social or cultural environments, for example, reading or solving problems.[1] Long-term memory is believed to process information unconsciously, has a virtually limitless capacity, and stores processed information in the form of schemata.

Intrinsic cognitive load is the load imposed on working memory by the nature and structure of the subject taught and is considered to be manageable but not amenable to change. The main source of intrinsic cognitive load is the element interactivity of the subject, which is determined by the learners' expertise and the number of elements of a task or subject that can be successfully learned without needing to refer to their relation with other elements (Sweller, van Merriënboer, & Paas, 1998). The cognitive load produced by the instructional material is called extraneous cognitive load. Sweller (2010) attributed extraneous cognitive load to element interactivity of instructional material and methods. This formulation seems to propose element interactivity as a common basis for explaining both intrinsic and extraneous cognitive load.

CLT can be used to support learning by freeing up memory resources devoted to processing extraneous cognitive load in order to reallocate them, as germane resources, to processing intrinsic cognitive load; this can be achieved through instructional design (Caspersen & Bennedsen, 2007; Sweller et al., 2011). A number of techniques that have been demonstrated to aid learning through the application of CLT in various disciplines include (a) the goal-free effect, (b) the worked-example effect, (c) the completion problem effect, (d) the split-attention effect, (e) the modality effect, (f) the redundancy effect, and (g) the expertise reversal effect (Morrison, 2015, 2017; Morrison, Decker, & Margulieux, 2016; Paas & van Merriënboer, 1994; Sweller et al., 1998; van Merriënboer & Sweller, 2005; van Mierlo, Jarodzka, Kirschner, & Kirschner, 2011).

Design of Koios

Motivation for the creation of Koios. Many modern educational programming environments, like Scratch (Resnick et al., 2009), Alice (Bishop-Clark et al., 2007), and Greenfoot (Kölling, 2010), have been influenced (indirectly) by principles of constructivism, although constructivism is often not explicitly referenced as part of their design rationale. Based on CLT, other environments for novice programmers like CORT (Garner, 2009), ReadJava simulator (Williams, 2014), XLogoOnline (Hromkovič, Serafini, & Staub, 2017) were designed explicitly to reduce cognitive load on its users. Moons and De Backer (2013) combined principles of these two learning theories in the design of their interactive learning environment for introductory programming, and its evaluation showed that it

facilitated its users in understanding programming constructs most students find difficult.

There is a generation of modern educational programming tools, like Scratch or Alice, which are used for teaching and learning programming worldwide, targeting primarily an English-speaking audience with available versions of these tools in other languages, like Greek. However, we were motivated to design and implement our own programming tool especially tailored for Greek novice programmers by the following four reasons. First, because of the positive findings of Moons and De Backer (2013) and the fact that, at least to our knowledge, an educational programming tool designed based on principles of both these two theories has not been used for introductory programming in Greece. Second, based on CLT, we argue that extraneous cognitive load could be reduced by providing a less complex IDE (Mason, Cooper, Simon, & Wilks, 2015) and a smaller set of commands can be beneficial for users (Koulouri, Lauria, & Macredie, 2014). Third, we intended users to focus on learning the mechanics of programming (Williams, 2014) by providing a more transparent and detailed view of program execution, which is not supported by existing IDEs. Fourth, at the time of the study, educational programming environments that were popular elsewhere were not in use in Greece. This was also the reason for deciding to include two programming tools that were popular for programming instruction in Greece at the time of the study.

In addition to the two theories, the design process was also influenced by previous studies and educational programming software relevant to novice programming. Thus, a number of design choices was inspired by or based on recommendations that were proposed in the work of Pane and Myers (1996), Green and Petre (1996), McIver (2000), Kelleher and Pausch (2005), Mannila and de Raadt (2006), and Sorva, Karavirta, and Malmi (2013). In addition, we were inspired by educational programming tools such as BACCII (Calloni et al., 1997), FLINT (Ziegler & Crews, 1999), jGRASP (Hendrix et al., 2004), VIP (Virtanen, Lahtinen, & Järvinen, 2005), RAPTOR (Carlisle, Wilson, Humphries, & Hadfield, 2005), ALVIS Live! (Hundhausen & Brown, 2007; Hundhausen et al., 2009), Scratch (Resnick et al., 2009), Alice (Bishop-Clark et al., 2007), JEliot (Ben-Ari et al., 2011), and Ville (Rajala et al., 2008). Next, we present the most important choices and heuristics used for designing Koios.

Linguistic attributes of Koios match users' natural language. Koios was designed for novice programmers who study at Greek educational institutes. Therefore, Koios' programming commands as well as IDE were developed to match the target users' natural language, namely Greek,[2] with the aim of reducing extraneous cognitive load (Bouvier et al., 2016). In addition, Roussel, Joulia, Tricot, and Sweller (2017) found that presenting material in a foreign language decreased language and content learning.

Providing a minimum set of commands for novices. The concepts supported by Koios were determined by (a) the target users of Koios, (b) computer science curricula of educational institutes, and (c) the aim to reduce extraneous cognitive load. The target audience of Koios is novice programmers and thus all the basic programming concepts should be supported, namely constant, variable, array, arithmetic operators, relational operators, logical operators, input-statement, output-statement, assignment-statement, conditional statement, iteration-statement, function, and procedure.

Creating programs visually. Another design decision was that the cognitive load imposed by the syntax of commands should be minimized in order to allow novices to focus on programming concepts during program creation (Gaspar, Langevin, & Boyer, 2008; Grandell, Peltomäki, & Salakoski, 2005; Lye & Koh, 2014; McIver, 2000; Myers, 1990; Scarr, Cockburn, Gutwin, & Quinn, 2011; Scott, 2010). Thus, Koios users can create their programs only in a graphical or visual way (Petre, Blackwell, & Green, 1998). Typically, programs created in an introductory course are small and therefore, creating small programs visually can be a powerful method, especially in programming tools for novices. Moreover, visual tools have been found to promote transfer-of-learning when users move to textual PLs (Stefik & Hanenberg, 2014). A list of icons represents the available programming commands because images seem to facilitate the construction of schemata (Tudoreanu & Kraemer, 2008), which according to constructivism, takes place during knowledge acquisition. A programming command can be added into a program by dragging-and-dropping or copying-and-pasting its icon into the program (Lee & Ko, 2011).

Dropping (or pasting) a command's icon in programs initiates a dialog session between users and Koios via dialog windows (Hsu, 2003), during which, they are guided through the creation of this command and provide all the necessary parameters regarding it. Thus, the detection of syntax errors during the creation of each command can be facilitated, and immediate feedback can be given to users (Carroll & Carrithers, 1984; Grandell et al., 2005) via error messages. Hence, the functionality of an interpreter is simulated, which is considered beneficial for novice programmers (Gaspar et al., 2008). We acknowledge that the use of dialogs could become frustrating for experienced users, but the benefit of having an always syntax-error-free program is more important than a potential inconvenience. From a constructivist viewpoint, users are able to construct their new programming knowledge, given that they are actively engaged in the creation of commands through a potentially repetitive process and supported by immediate feedback that could expose any misconceptions. Most important, a command is added in the program only when its parameters are specified correctly, which guarantees that programs are always without syntax errors and can be executed at any time. This process could also reduce extraneous cognitive load, because users do not need to recall the syntactic rules and required

parameters for correctly creating a command (Gaspar et al., 2008; Scott, 2010). Because Koios guides novices through the creation of each command, it may enable them to focus on programming concepts and their functionality.

Supporting graphical and textual notations. A basic consideration throughout the design process was that Koios should be a programming tool for the initial stages of programming instruction and, therefore, should provide features that would render the transition to professional PLs as smooth as possible (Hewett, 2005; Stefik & Hanenberg, 2014). This is the reason that programs can be viewed in a graphical or a textual way (Chilana et al., 2015; Pane, Myers, & Ratanamahatana, 2001; Petre et al., 1998). Users can switch at any time between views and familiarize themselves with both notations as well as the association between them. This frequent-switch functionality facilitates users not only make clearer associations between graphical and textual forms of commands and increase intrinsic cognitive load but also avoid increasing harmful cognitive load, as would be the case of a single switch between views before program execution. Programming icons do represent parameterized pieces of code in Koios' PL. However, this code is hidden from users when they are interacting with the programming icons. This approach underlies the distinction between designing a program and writing code (van Merriënboer, 1990).The textual view provides a *traditional* textual code that follows a specific set of syntactic and semantic rules (Vasilopoulos, 2014) and is created automatically by the graphical representation of the program.

Providing natural syntax and unambiguous semantics of commands. As the syntax and semantics of PLs are a source of mistakes for novices (Brown & Altadmri, 2017), the design aim was to produce a syntax with Greek keywords that feels natural in the Greek language while conforming to a certain level of formality. A trade-off here was that in order to facilitate users' transition to and familiarize them with professional PLs, Koios' syntax (Vasilopoulos, 2014) deliberately includes a small number of widely used programming words or symbols that we acknowledge that could be considered as less natural or intuitive for novices (Stefik & Siebert, 2013). Because the textual form of programs is created automatically and syntactic rules do not need to be remembered, the phrasing of commands in textual view can be verbose and more descriptive than typical programming commands. This, in turn, can make the textual form of programs more readable and informative and reduce extraneous cognitive load. Koios' semantics of commands were designed to convey their programming meaning and to be as unambiguous as possible, while reducing extraneous cognitive load. Thus, the auto-generated code follows conventional rules of indentation, while different combination of colors and font styles are used to distinguish different programming constructs.

Providing information during program execution. Programs are executed in a new window and shown in textual form. Koios users create their programs in a

graphical way and, thus, they are required to pay much attention to this form of programming. Involvement with the textual form of programs is important for any novice programmer as well, because at present, the dominant way of creating programs is by using text, and a visual representation of large programs is still inefficient. The textual form of programs supported by Koios can be more readable and informative—due to the verbose and more descriptive phrasing of commands in textual view—than the textual form of programs supported by other programming tools. Therefore, presenting programs in textual form during execution draws users' attention to this form as well. This feature, however, would be useful if users have already familiarized themselves with the textual code, during program creation. Otherwise, such a switch right before program execution could increase extraneous load.

To make the execution of programs more comprehensible and reduce working-memory load, each command is highlighted while it is being executed, and explanatory and commentary messages are shown regarding the function of the command (du Boulay, O'Shea, & Monk, 1999; Petre et al., 1998). Two features that further increase users' active involvement, which in turn, according to constructivism, can facilitate knowledge construction are (a) buttons that can start, pause, and terminate as well as control the speed of programs' execution are available and (b) data input during execution is performed via dialog windows (Hsu, 2003).

Two important additional design choices are (a) the programming paradigm supported by Koios and (b) the provision for comments in programs. The two dominant paradigms are the procedural and the object-oriented (OO) paradigm. Despite the ongoing debate on the appropriateness of each paradigm (Milne & Rowe, 2002; Wiedenbeck, Ramalingam, Sarasamma, & Corritore, 1999), the first design choice was to select the procedural paradigm as more appropriate, based on the set of programming constructs that are supported by Koios. This set can be supported by both paradigms, but it does not contain or require any OO concepts, as this would introduce unnecessary programming concepts, which, in turn, would increase extraneous cognitive load. Besides the increased complexity and possible *unnatural programming logic* for novices introduced by the OO paradigm (Koulouri et al., 2014; Rist, 1996), the procedural paradigm was considered more appropriate for introductory programming for the following three reasons. First, it seems that the OO paradigm is not more *natural*, easier, or powerful than the procedural paradigm as hypothesized (Green, 1990; Wiedenbeck et al., 1999); second, OO concepts are difficult for novices to understand (Milne & Rowe, 2002); and, third, novices need less time to familiarize themselves with the procedural paradigm (McCracken et al., 2001).

The second design choice was that the prototype of Koios does not need to support comments. Even though the inclusion of comments in programs is recognized as a helpful source in reading and understanding code, and a good programming habit (Raskin, 2005), the findings presented by Barr, Holden,

Phillips, and Greening (1999) and Grandell et al. (2005) suggest that the inclusion of comments by novices may yield undesired results. A number of students in the study of Barr et al. (1999) inserted comments that were similar to the code they had written, and in other cases, comments did not explain or were not relevant to the code. Grandell et al. (2005) reported that more than 65% of the participants in their study did not include comments in their code and that the quality of the comments did not always match the quality of the program. They also reported that reasons for omitting comments could be that students consider comments unnecessary for small programs, that the inclusion of comments delays the programming process and that properly conveying their programming ideas via comments is hard. This design decision perhaps seems controversial, but it does not suggest that comments in programs should altogether be removed from introductory courses, only that the programming tool does not necessarily need to support them. Further choices and decisions regarding the development and implementation of Koios are discussed next.

Implementation of Koios

The primary aim of Koios' design and development was producing a programming environment that could serve as an efficient tool for improving the teaching and learning of introductory programming in Greece. However, a secondary aim was that this tool could also serve as a stepping-stone for the transition of Greek novices to more advanced and powerful professional PLs. We invite our readers to try the Greek or English version of Koios, available on line at https://github.com/koiosVPT/koiosvpt_en.

The Koios programming environment was developed in Java, using the NetBeans IDE—Versions 6.5 and 6.7—and Java Development Kit 6, 7, and 8. Koios was developed in Java because it is hardware independent and platform independent (Gosling & McGilton, 1996). This way, hardware dependence is avoided, a *deadly sin* of PLs for novices (McIver & Conway, 1996).

Koios' user-interface. Koios provides a simple and easy user-interface for novice programmers, avoiding unnecessary exposure to more complex features. Furthermore, Koios provides two view modes for initial and advanced states of introductory programming, respectively.

As shown in Figure 1, the user-interface of Koios in program creation mode displays a menu bar, a toolbar, a status bar, and three main windows: (a) the program window (left), (b) the programming commands window (top right), and (c) the properties window (bottom right). The programming commands window contains all the available programming commands, grouped in six categories. The implemented programming commands-items *in Advanced mode* are constants, variables, arrays, write, read, assign, call, if-statement, for-statement, while-statement, do-while-statement, return procedures, and functions. The

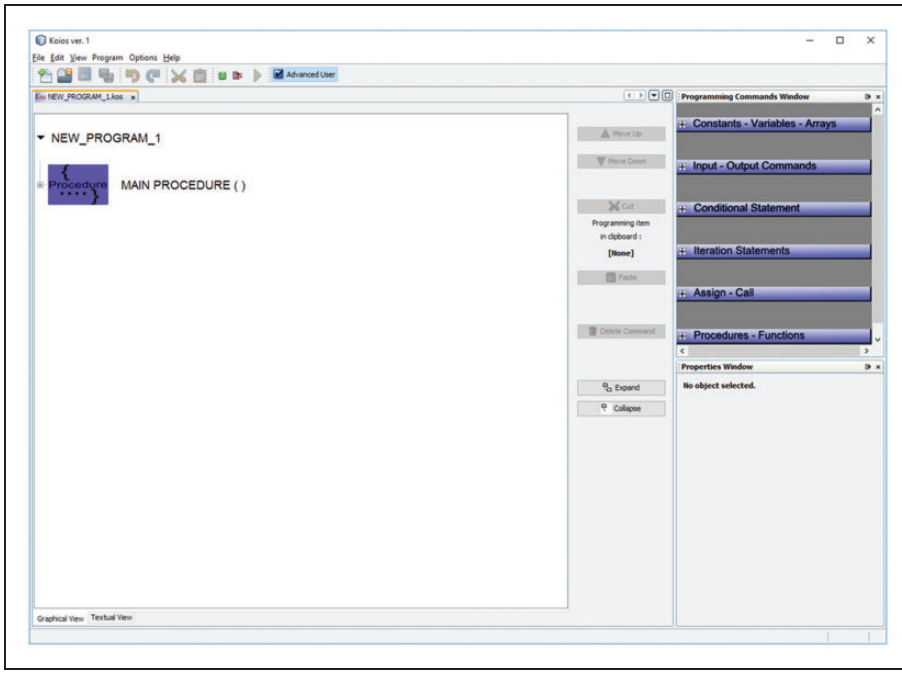

Figure 1. The main window of Koios programming environment.

Basic mode provides only the following programming commands-items: variables, write, read, assign, call, if-statement, for-statement, and procedures. Each programming command is represented by an icon; the available programming commands or items in *Advanced mode* and their grouping can be shown in Figure 2.

Programming items are represented as icons-nodes and are associated with brief text, which reads programming items' most important properties. The aim of this is not only to help users to identify different programming items that have the same icon but also to prevent them from splitting their attention between different parts of the user-interface by providing critical on-time information regarding items' properties (Shaffer, Doube, & Tuovinen, 2003). The nodes of programs are organized in a tree structure, which resembles the structure used for the representation of files and folders by the majority of operating systems. This choice was based on the following four reasons. First, it is possible that this view can convey the concept that there are two categories of nodes or programming items: (a) compound statements, which are represented by nodes or programming items that can *include* one or more nodes, like the if-statement and for-statements and (b) simple statements, which are represented by nodes or

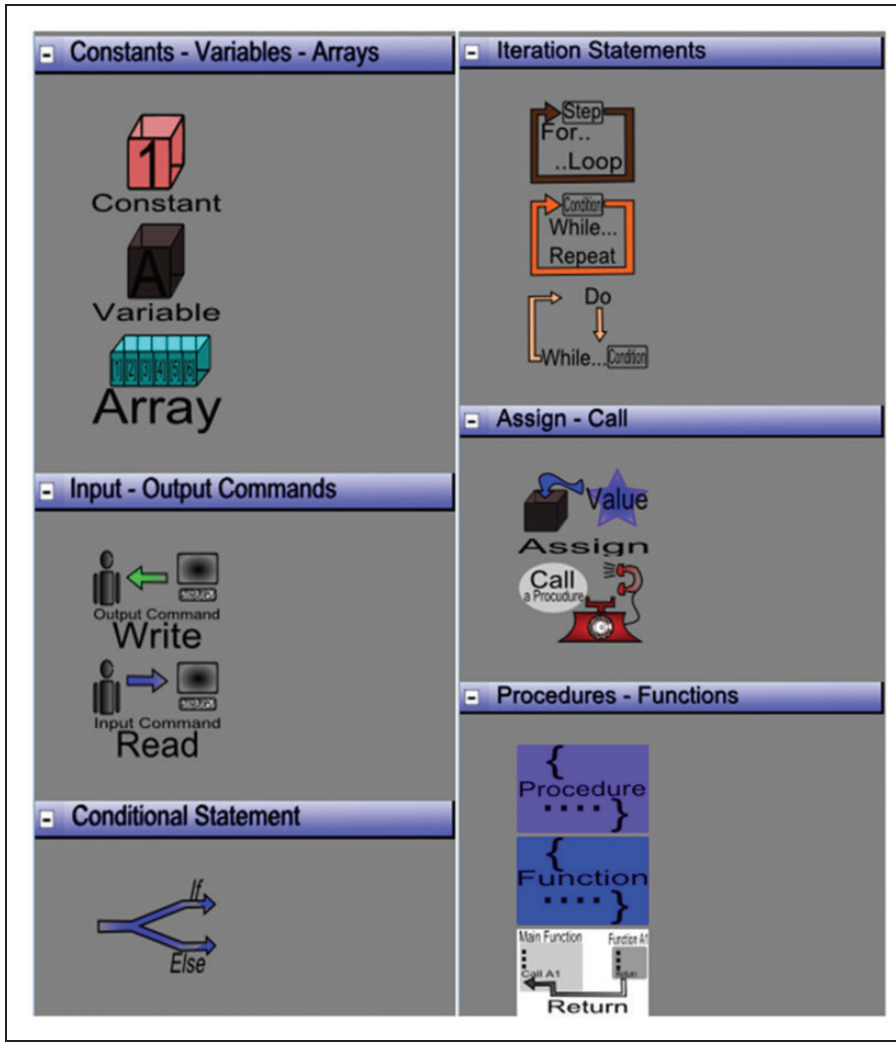

Figure 2. Icons of programming items.

programming items that cannot, like an output statement. Second, according to constructivism, previous experiences can be used to construct new knowledge and thus, users who may be already familiar with the tree view from their interaction with computers' operating system could better grasp the parallel. Even if this is not the case, this representation could be a good metaphor for distinguishing the functionality of these two categories. Third, this view can provide a good mechanism for showing or hiding the nodes or programming items within

a node or item, such as the commands within a for-statement and thus, the size of the visible program can be adjusted. Fourth similar showing or hiding techniques are used by modern professional textual PLs and IDEs.

New commands or items are added to a program by selecting an icon from the programming commands window and dragging-and-dropping or copying-and-pasting it to a program. To prevent errors, only valid commands are allowed to be entered in programs, for example, the icon of a return statement can be added in functions but not in procedures. When a new item is dropped or pasted in the program, a dialog between Koios and a user is triggered to guide them through the creation of the command and determine its properties. Before the user moves away from a dialog window, the properties entered so far are checked. If a property is not entered correctly, then an error message appears in a pop-up dialog window. When all the properties have been entered correctly, a new icon is created and a message appears informing the user that this programming item has been successfully added in the program. Hence, only items without errors are entered in the program; therefore, the program is always in a syntax-error-free state.

Every time an action is performed, a message appears on the left side status bar that describes this action, while on the right side of the status bar, a balloon appears that provides information about the status of programs. Available actions are adding, deleting, cutting and pasting, and moving up or down programming items.

Code generation and error check. The graphical and textual view of a program can be shown in Figure 3. Figure 3(b) shows the text code that is automatically generated from the graphical view of the program (Figure 3(a)). Different combination of colors and font styles are used for formatting the text in order to highlight the different programming constructs.

Koios provides a check command, which performs three actions on the current program. First, it reorders the nodes of constants, variables, and arrays that are declared and moves them to the beginning of the procedure or function. This action guarantees that every item is declared before it is used. Second, it checks whether any constants, variables, or arrays are declared but not used and any variables or array elements are used without initialization. In the first case, the name of the item is reported, while in the second case, the name of the item as well as the statement(s) that include(s) it are reported. Third, it detects any nonsyntax errors, for example, functions without return statements, except for dynamic semantic (run-time) errors, for example, accessing an array element beyond the size of the array, which can only be detected during run-time. From a pedagogical viewpoint, users are advised to run the check command during program creation and before compiling the program. However, if this action is omitted the compile command automatically first runs the check command before actual compilation of the program. When the check or compile

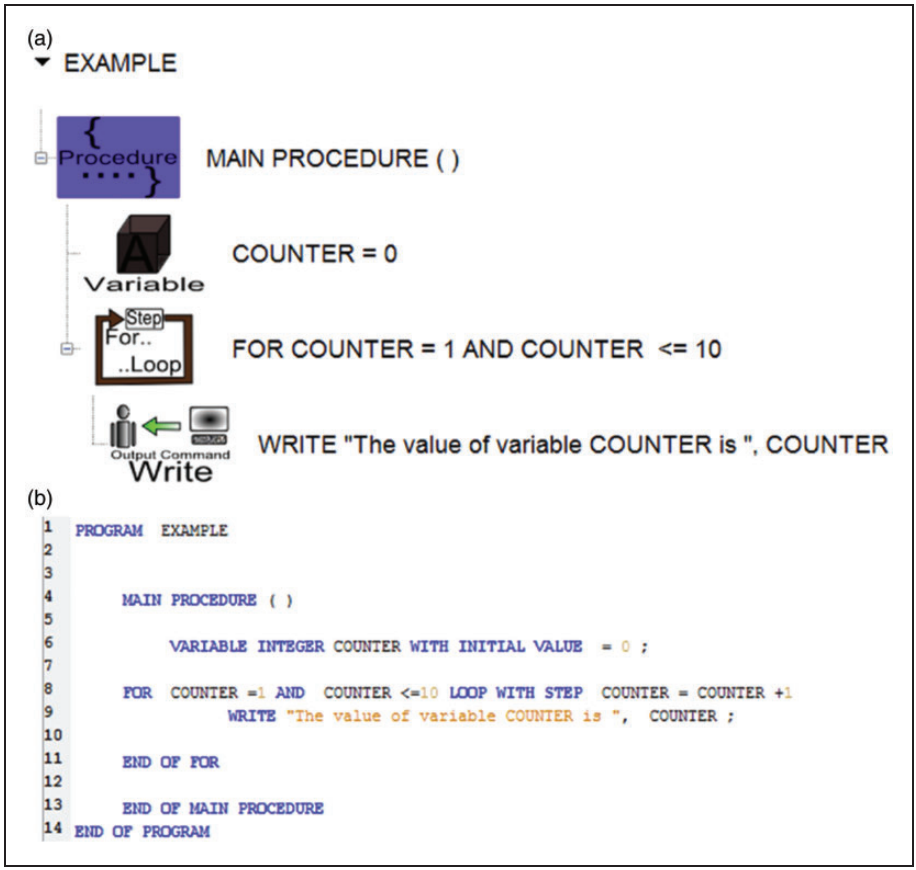

Figure 3. The (a) graphical and (b) textual view of a program.

commands are executed, the balloon in the status bar informs users whether warnings or errors exist or not.

Koios' complier. Deciding which compiler would be appropriate is a matter that comes with a number of technical considerations. Since Koios uses the Java Virtual Machine, the solution was to transcompile the Koios programs to equivalent Java programs. These programs can, in turn, be executed within the Koios programming environment. It should be noted that the equivalent Java programs are carefully transcompiled to produce as few Java exceptions (errors) as possible. However, if such an exception occurs, it is handled transparently to Koios users. An important feature of Koios is that it provides the Export Java Program command, which can produce the equivalent Java version

of the Koios programs. This feature can be useful for a smooth transition from an *introductory* PL to a professional one or if the equivalent Java code is needed.

Program execution mode. Once a program contains no errors, it can be executed using the run command. This action changes Koios' mode from program creation to program execution, used for tracing program execution. At the top of the window, there are control buttons for starting, pausing, and stopping the execution and a slide bar for adjusting its speed. When a statement is being executed, it is highlighted in yellow, while in the case of a compound statement, for example a for-statement, the end statement is highlighted in orange (see Figure 4). The output of running programs is shown in the output window of the program execution mode.

An important and unique feature of Koios is the comments or description window. This window provides messages with overall information about the statement that is being executed at the time. This information includes a brief description of the statement's properties, how this statements works, what the result of its execution is, and how it affects the control flow of the program (if applicable). Thus, through the play or pause buttons and these messages users can be supported in better understanding while tracing code execution.

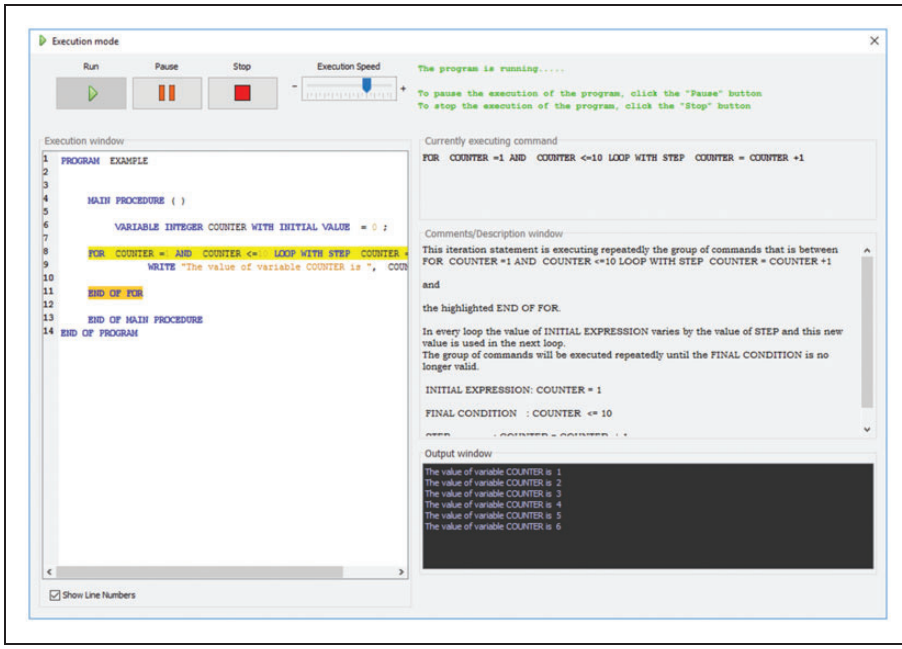

Figure 4. A snapshot of the execution of a program.

These features were developed because, according to constructivist principles, users—based on their personal learning needs—can individually and actively construct their knowledge structures of program execution at their own pace and with minimum help from instructors and promote explanatory learning under proper guidance with always-available on-time information.

Empirical Study and Evaluation of Koios

Method

The method followed for evaluating Koios programming tool (Vasilopoulos, 2014) is by comparing their impact with that of other programming tools on users (Stefik & Hanenberg, 2014; Vogts, Calitz, & Greyling, 2008). The comparison of a new programming tool with a widely used one can provide evidence for its effectiveness and contribution.

At the time of this study, there were two popular programming environments in Greek secondary-level education that were used for teaching introductory programming: Glossomatheia (Nikolaidhs, 2008) and MicroworldsPro (LCSI, 2008). Glossomatheia is a Greek PL and IDE based on Pascal, while MicroworldsPro is an English LOGO-based programming environment available in the Greek language. For both programming tools, various versions have been developed, which renders these rather *mature* programming tools. In fact, both tools have recently released new versions, namely Microworlds EX and Glossomatheia 9.2.2. Koios was evaluated in comparison with Glossomatheia and MicroworldsPro through an empirical study that took place within an actual secondary-school learning environment and was based on the following research design. Table 1 presents differences between the three programming tools across basic features of PLs and IDEs such as syntax and semantics, use of visual techniques, set of instructions, level of comprehensible error messages, and level of interaction with the IDE.

Table 1 shows that Koios provides more natural syntax and unambiguous semantics than Glossomatheia and MicroworldsPro that provide Pascal-like and Logo-like syntax with Greek keywords, respectively. Moreover, Koios, compared with the medium level supported by Glossomatheia and MicroworldsPro, provides a high level of visual techniques. All programing tools provide a small set of instructions, but Koios' error messages are designed to be more comprehensible and informative than the error messages provided by Glossomatheia and MicroworldsPro. Finally, Koios' IDE maintains a higher level of interaction with its users during program creations and execution than the other two programming tools.

Research hypothesis. Koios' design was inspired by principles of both constructivism and CLT, but the design of Glossomatheia is not explicitly based on either

Table 1. Differences Across Features Between the Three Programming Tools Used in the Quasi-Experimental Design.

	Features				
Programming tool	Syntax and semantics	Use of visual techniques	Set of instructions	Level of comprehensible error messages	Level of interaction with the integrated development environment
Koios	Natural syntax and unambiguous semantics	High level	Small	High	High
Glossomatheia	Pascal-like	Medium level	Small	Medium	Medium
Microworlds Pro	Logo-like	Medium level	Small	Low	Medium

of these. However, MicroworldsPro, as a programming tool based on LOGO (Papert, 1980), could be considered to have been influenced by constructivism. Therefore, the following research hypothesis was tested: Novice students who use Koios develop a higher level of programming competence than novice students who use Glossomatheia and MicroworldsPro.

Design. A quasi-experimental design was used, because this study took place with actual school students as participants and, due to administrative reasons, a random allocation of participants was not possible (Lye & Koh, 2014; Stefik & Hanenberg, 2014). The independent variable of this design was the programming tool, and the dependent variable was the level of programming performance of novice programmers.

Participants. Statistical power analysis revealed that a total of 66 Greek secondary-school pupils were required in order to detect a large effect size ($\eta^2 = .138$) with a significance level of .05 and a statistical power of .80. The 70 participants—37 male students and 33 female students (mean age = 14)—that took part in this study were third-grade secondary-school students with no previous formal instruction in programming. Three whole school-classes were allocated by the school administration as the three quasi-experimental groups, and each one consisted of 23 (Koios group), 25 (Glossomatheia group), and 22 (MicroworldsPro group) participants-students. Participants were informed about the study, and consent forms were signed by their parent(s) or guardian(s) before the start of the study. Participants had no previous programming experience nor had they used a programming tool before.

Procedure. Each group used the corresponding programming tool for its introductory programming course for nine lessons (one per week) and had the same teacher (first author). The programming concepts, teaching method, classroom examples, homework, and teaching material were the same for all experimental groups. The only necessary, though slight, variation was the teaching material for each of the three groups in order to appropriately present the corresponding programming tool. Programming performance of participants was assessed with a practical test at the end of the course, as this type of task has been found to be provide an accurate assessment of programming performance (Koulouri et al., 2014, p. 14).

Material. Participants were asked to create three programs in computer lab with the programming tool they had been using during their programming course. The three programs are presented in Appendix A.

Data analysis. For the analysis of the practical-test scores, correlation analysis, simple regression, and analysis of covariance ran as hierarchical multiple regression (HMR; Field, 2017; Pedhazur & Pedhazur Schmelkin, 1991) were employed. These techniques were used for testing the effect of programming tool on the level of programming skills. The programs that students produced were scored by the first author using an assessment scheme (see Appendix B), and the minimum and maximum score of the practical test was 0 and 30, respectively. The same scheme was used by an external marker for marking the programs of the practical test in order to analyze reliability of marking.

Gender, assessments of homework, grades from other relevant subjects (e.g., mathematics and physics), and grade point average (GPA) were used as candidate covariates. However, only participant's GPA correlated significantly with the end-test scores, $r(66) = 0.51$, $p < .001$ and therefore was the only covariate used in the analysis.

Because Koios programs are always syntax-error-free, the data collected regarding syntax test-items were analyzed separately from the nonsyntax items. This separation removed the potential bias that the syntax items could introduce if all items would be analyzed together.

Results

The reliability of the syntax and nonsyntax items of the end test was high in both cases, Cronbach's alpha = .90 and Cronbach's alpha = .94, respectively. The reliability of marking, for which a subset of 16 programs of the end test was used, was high as well, $r(14) = .99$, $p < .001$. Descriptive statistics for the syntax and nonsyntax scores of the end test are presented in Tables 2 and 3, respectively.

As shown in Table 2, the mean values for the syntax items of the Glossomatheia and the MicroworldsPro groups were 1.49 and 1.40 out of 6,

Table 2. Descriptive Statistics for Syntax Scores of the Practical Test.

Group	*n*	Mean	*SD*	*SE*	Cohen's *d*
Koios	23	2.68	1.78	0.37	
Glossomatheia	25	1.49	1.38	0.28	0.75
MicroworldsPro	22	1.40	1.57	0.34	0.76
Total	70	1.85	1.67	0.20	

Table 3. Descriptive Statistics for Nonsyntax Scores of the Practical Test.

Group	*n*	Mean	*SD*	*SE*	Cohen's *d*
Koios	23	11.54	7.94	1.66	
Glossomatheia	25	6.90	6.00	1.20	0.66
MicroworldsPro	22	5.94	6.73	1.44	0.77
Total	70	8.12	7.23	0.86	

respectively, while the mean score of the Koios group was 2.68. Table 3 shows that the mean values for the nonsyntax items of the Glossomatheia and the MicroworldsPro groups were 6.90 and 5.94 out of 24, respectively, while the mean score of the Koios group was 11.54; almost twice the mean score of each of the other two groups. According to Cohen's (1988) conventions, the effect size observed in the mean difference between the Koios and the Glossomatheia groups, as well as between the Koios and the MicroworldsPro groups was medium to large for both syntax and nonsyntax items. The results of HMR of the syntax end-test scores are presented in Table 4.

In the first regression model, GPA was the single predictor and explained 21% of variance in the scores, $R^2 = .21$, $F(1, 68) = 18.49$, $p < .001$, which was statistically significant. In the second model, programming tool was introduced in the regression model, and accounted for a significant additional 21% of variance in the scores, $\Delta R^2 = .21$, $F_{change}(2, 66) = 11.38$, $p < .001$, and a total 42% of variance was explained by the two predictors, $R^2 = .42$, $F(3, 66) = 15.63$, $p < .001$. GPA remained a significant predictor, $t(66) = 5.74$. Finally, with GPA held constant, Koios users performed significantly better, with on average 1.19 (Table 2) points higher on the end test than Glossomatheia users, $t(66) = -4.11$, $p < .001$; they also performed significantly better, with on average 1.28 (Table 2) points higher on the end test than MicroworldsPro users, $t(66) = -4.20$, $p < .001$. The results of HMR of the nonsyntax end-test scores are presented in Table 5.

Again, in the first regression model, GPA was a significant predictor and explained 23% of variance in the scores, $R^2 = .23$, $F(1, 68) = 20.35$, $p < .001$.

Table 4. Results of Regression Analysis for the Syntax Scores of the Practical Test.

Predictor	*b*	*SE*	B	*t*
Model 1				
Constant	−3.72	1.30		**−2.89
Participant's GPA	0.33	0.08	0.47	***4.30
Model 2				
Constant	−3.70	1.14		**−3.23
Participant's GPA	0.39	0.07	0.55	***5.74
Glossomatheia vs. Koios	−1.57	0.38	−0.46	***−4.11
MicroworldsPro vs. Koios	−1.65	0.39	−0.46	***−4.20

Note. GPA = grade point average; $R^2 = .21$ for Model 1; $\Delta R^2 = .21$ for Model 2 ($p < .001$).
**$p < .01$. ***$p < .001$.

Table 5. Results of Regression Analysis for the Nonsyntax Scores of the Practical Test.

Predictor	*b*	*SE*	B	*t*
Model 1				
Constant	−17.00	5.62		**−3.02
Participant's GPA	1.49	0.33	0.48	***4.51
Model 2				
Constant	−16.92	4.95		**−3.42
Participant's GPA	1.75	0.27	0.57	***5.92
Glossomatheia vs. Koios	−6.32	1.65	−0.42	***−3.83
MicroworldsPro vs. Koios	−7.25	1.70	−0.47	***−4.26

Note. GPA = grade point average; $R^2 = .23$ for Model 1; $\Delta R^2 = .19$ for Model 2 ($p < .001$).
**$p < .01$. ***$p < .001$.

In the second model, programming tool explained a significant additional 19% of variance in the scores, $\Delta R^2 = .19$, $F_{change}(2, 66) = 10.89$, $p < .001$, while a total of 42% of variance was explained by the two predictors, $R^2 = .42$, $F(3, 66) = 16.00$, $p < .001$. GPA was also a significant predictor, $t(66) = 5.92$. Finally, with GPA held constant, Koios users performed significantly better, with on average 4.64 points (Table 3) higher on the end test than Glossomatheia users, $t(66) = -3.83$, $p < .001$; they also performed significantly better, with on average 5.60 points (Table 3) higher on the end test than MicroworldsPro users, $t(66) = -4.26$, $p < .001$.

The bias of the regression model was also tested. The assumptions of homoscedasticity, linear relationship between outcome and predictors, and

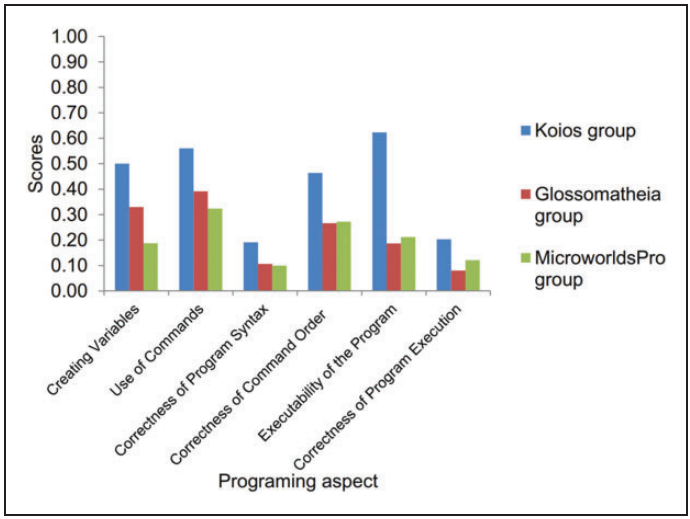

Figure 5. Mean scores for each programming aspect per group.

independence of residuals were met. In addition, variance inflation factor and tolerance values revealed that no multicollinearity existed in analyzed data sets. Finally, the assumption of homogeneity of regression slopes was also met (Field, 2017).

To further investigate the aspects in which Koios users achieved higher scores than users of MicroworldsPro and Glossomatheia, an in-depth analysis of the end-test scores was performed. Figure 5 presents each group's mean scores with respect to aspects of creating and executing a program.

The six aspects considered in this in-depth analysis were as follows: (a) creation of variables, (b) overall use of commands (input-statement, output-statement, assignment-statement, iteration-statement, conditional statement, and creating and calling a procedure), (c) correctness of program syntax, (d) correctness of command order, (e) executability of the program, and (f) correctness of program execution. The grades achieved by the students of each group were averaged over the three programs for each aspect. Hence, the scores for each aspect ranged from zero (minimum) to one (maximum). As can be seen in Figure 5, students who used Koios to create the programs of the practical test seemed to have performed better than the users of the other two programming tools in all aspects. The values of mean, standard deviation, and Cohen's *d* (1988) of the three groups for the six programming aspects are presented in Table 6.

More specifically, the mean for the Koios group was higher than that for the Glossomatheia and the MicroworldsPro groups on the following programming aspects: creation of variables, use of commands, correctness of program syntax, correctness of command order, executability of the program, and correctness of

Table 6. Descriptive Statistics for the Six Programming Aspects.

	Group							
	Koios		Glossomatheia			MicroworldsPro		
Programming aspect	*Mean*	*SD*	*Mean*	*SD*	Cohen's *d*	*Mean*	*SD*	Cohen's *d*
Creating variables	0.50	0.35	0.33	0.35	0.49	0.19	0.30	0.96
Use of commands	0.56	0.57	0.39	0.48	0.32	0.32	0.46	0.46
Correctness of program syntax	0.19	0.18	0.11	0.14	0.53	0.10	0.15	0.56
Correctness of command order	0.46	0.50	0.27	0.33	0.47	0.27	0.39	0.43
Executability of the program	0.62	0.48	0.19	0.30	1.09	0.21	0.38	0.95
Correctness of program execution	0.20	0.39	0.08	0.15	0.43	0.12	0.32	0.23

program execution (Table 6). As shown in Table 6 and according to Cohen's (1988) conventions, the effect sizes observed for the difference of the mean scores for the different programming aspects between the Koios and the Glossomatheia groups and between the Koios and the MicroworldsPro groups each varied from small to very large. Therefore, although an advantage of Koios was observed for each of the six aspects and not confined to a subset, the largest advantage by far was in program correctness.

Discussion

We discuss the main contribution of the work that is presented in this article and potential extensions. Finally, we discuss limitations of our work and future research.

Koios Contribution to Research and Knowledge

The design and development of this new Greek programming tool, Koios, was motivated by our intention to facilitate Greek novice programmers' learning in their introductory programming course and, thereby, to contribute to knowledge. Therefore, Koios' design and implementation was guided by heuristics and rules of thumb that were derived from the combined principles of constructivism and CLT. Koios includes a PL and IDE in users' native language that provides a minimum set of instructions for novices, which seems to have more natural-language-like syntax and semantics. Moreover, Koios supports visual creation of programs, both graphical and textual notations, constant guidance,

feedback, and comprehensible error messages during program creation. These guarantee that programs in Koios are always in syntax-error-free state. In addition, Koios supports visualization and explanatory messages regarding program execution.

The combination of all these features supported by Koios constitute a major advantage for our tool against other Greek programming tools that only consider part of these features. Moreover, Koios' design and implementation followed a user-focused approach, as developing a software tool that helps and supports its users in efficiently accomplishing tasks, which is particularly important especially for complex and challenging tasks like programming. Thus, Koios' characteristics (e.g., textual code automatic generation, error prevention, informative feedback, and constant guidance) could support its users in overcoming common difficulties novices face and enable them to focus on the task of programming.

A second contribution of this research is establishing that Koios can effectively facilitate Greek novices in program creation. First, Koios significantly facilitated its users in creating programs, even when they created them on their own and without guidance or help from teachers. More specifically, the results of the end test revealed a significant effect of the quasi-experimentally controlled factor programming tool (Koios vs. MicroworldsPro and Glossomatheia) on end-test scores. In particular, Koios users performed significantly better regarding both syntax and nonsyntax items of the practical test than the users of the two other tools. Although a significant performance regarding syntax items from Koios users was anticipated because Koios does not allow syntax mistakes, the significant test results for nonsyntax items more generally reveal the effectiveness of Koios as a programming tool for Greek novices. Moreover, Koios users appear to perform better than users of *standard* software in the following six programming aspects: creation of variables, overall use of commands, correctness of syntax, correctness of command order, executability of the program, and correctness of program execution. Second, GPA was a significant independent predictor of students' performance in the end test. Therefore, although novices' overall academic ability predicts programming competence, programming tool explains additional variance in competence. Furthermore, because gender did not correlate significantly with the end-scores, this study corroborated the finding from previous studies that gender does not affect programming performance (Kahler, 2002; Lau & Yuen, 2011; Linn, 1985; Pillay & Jugoo, 2005).

These results are explained by the fact that the rules of thumb and heuristics that guided Koios' design and implementation were primarily based on the combination of principles of constructivism and CLT, while the design of Glossomatheia and MicroworldsPro is not explicitly influenced, at least not to the extent that Koios is, by these two theories. Designing programming software based on principles of and heuristics inspired by constructivism and CLT can

produce a programming tool capable of (a) reducing extraneous cognitive load, (b) providing a proper management of intrinsic cognitive load, (c) reallocating more working-memory resources as germane resources, (d) facilitating the understanding of programming concepts and reducing possible misconceptions, (e) actively engaging users in program creation and execution, (f) providing an easier and more comprehensible way of program creation, and (g) presenting a clearer view and supporting a better understanding of program execution. Thus, principles and heuristics that are based on these two theories should be used to inform the design of an effective and supportive learning environment for novice programmers.

Limitations and Future Work

Limitations. The main source of bias for this study could be the fact that the first author was the person who designed and developed Koios, was the teacher in three quasi-experimental groups and collected the end-test data. The evident conflict of interests and potential bias could genuinely affect the validity of this study. To moderate this bias, the marking process was made more objective by producing a marking scheme for the practical test and recruiting an external marker to mark a sample of students' responses. The reliability of marking of the practical test was calculated using external marker's assessment and was found to be high. Another limitation of this study was the use of a quasi-experimental design instead of an experimental design (Stefik & Hanenberg, 2014), which could not be used for constraints imposed by school administration.

Future work. In future research, we look forward to further evaluate Koios in comparison to Greek versions of modern and popular programming tools, like Scratch or Alice (Chih-Kai, 2014). We would also like to employ measurements that could reveal a more in-depth knowledge of programming concepts acquired by using these specific programming tools.

An important aspect for further improvement of Koios' design and implementation could be the on-going refinement of the syntax and semantics supported by Koios based on new rigorous research findings, such as the ones reported by Stefik and Siebert (2013). Furthermore, proven techniques of CLT (see Section Theoretical Background of Koios) could be a considerable source of inspiration in further improving Koios. Moreover, the messages and comments during execution could be improved taking into account findings reported by Koulouri et al. (2014), Stefik et al. (2011), and D'Mello, Graesser, and King (2010). These improvements could aim at highlighting explanatory comments' content provided by Koios during execution, making comments to be perceived as more natural by users, and help them in the debugging process. Another suggestion may involve displaying the graphical view together with the textual view during execution. This juxtaposition could further increase

comprehension and allow users to refer to and focus on a more compact (graphical) and a more detailed view (textual) of the code simultaneously, which is supposed to help programmers better understand programs (Green & Petre, 1996). Moreover, during execution users might create more associations between the graphical and the textual view of the code as well as between these and programs' functionality.

In addition, modifications in Koios' design and development could be made to further improve its educational impact. For example, additional modes could be provided with respect to the execution of programs. Hence, programs could be executed in a step-by-step mode, could allow users to return to a previous state in execution, and overall provide simple features of professional debuggers. Similarly, these modes could be used to increase the visibility of the notional machine (du Boulay et al., 1999) and, more particularly, expose the intermediate states and steps during program execution.

Conclusion

In conclusion, we have designed and implemented a new highly interactive and visual Greek programming tool, namely Koios, based on combined principles of constructivism and CLT that guided our design. The benefits of Koios over widely used programming tools for novices in Greece were empirically demonstrated. We look forward to the application and further development of this work in programming instruction.

Appendix A. End Test Used in Empirical Study

The end test consisted of the following items and took approximately 30 minutes to complete. Students were asked to create the following three programs with the help of the programming tool they had been using during the study.

Program 1 (10 marks)

Create a program using Koios or Glossomatheia or MicroworldsPro that requests from the user to give (input) a name to the string variable NAME and an integer number to integer variable TIMES. The program prints to the screen the value of NAME as many times as the value of TIMES.

Program 2 (10 marks)

Use Koios/Glossomatheia/MicroworldsPro to create a program with a procedure named OPP. Procedure OPP takes an integer input parameter X and prints to the screen the opposite number of X, namely—X. The main procedure requests from the user to give (input) an integer number to the integer variable NUMBER, and with the use of OPP prints to screen the opposite value of NUMBER.

Note: MicroworldsPro does not support the expression -X, to calculate the opposite of X. Therefore, the following hint was given to MicroworldsPro users only.

Hint: Calculate –X using the formula: $-X = -1 \times X$.

Program 3 (10 marks)

Create a program using Koios or Glossomatheia or MicroworldsPro that requests from the user to give (input) an integer number to the integer variable GRADE. If the value of GRADE is less than 10, then the string FAIL will appear on the screen, otherwise the string PASS will appear on the screen.

Appendix B. Marking Scheme for End Test

The following scheme was used in order to objectively assess the programs produced during the end test. Each produced program received a specific number of points for including the appropriate programming element(s) or appropriately incorporating particular programming properties. The mark of each program was the sum of the points received. The maximum number of points awarded for each programming element or property for the three programs is presented in Tables B1, B2, and B3.

Table B1. Marking Scheme for the First Program of End Test.

Elements and properties of Program 1		Points
Syntax items		
	Correct syntax of declaration of variables	0.25
	Correct syntax of output statement	0.50
	Correct syntax of input statement	0.50
	Correct syntax of iteration statement	0.75
	Total	2.00
Nonsyntax items		
	Declaration of variables	0.75
	Use of input statement	1.25
	Use of output statement	1.25
	Use of iteration statement	1.75
	Correct order of commands	1.00
	Program execution	1.00
	Correct program execution	1.00
	Total	8.00

Table B2. Marking Scheme for the Second Program of End Test.

Elements and properties of Program 2		Points
Syntax items		
	Correct syntax of declaration of variables	0.25
	Correct syntax of input statement	0.25
	Correct syntax of output statement	0.25
	Correct syntax of procedure and input parameter creation	0.625
	Correct syntax of procedure calling	0.625
	Total	2.00
Nonsyntax items		
	Declaration of variables	0.75
	Use of input statement	0.75
	Use of output statement	0.75
	Creation of procedure and input parameter	1.375
	Calling procedure	1.375
	Correct order of commands	1.00
	Program execution	1.00
	Correct program execution	1.00
	Total	8.00

Table B3. Marking Scheme for the Third Program of End Test.

Elements and properties of Program 3		Points
Syntax items		
	Correct syntax of declaration of variables	0.25
	Correct syntax of input statement	0.25
	Correct syntax of output statement	0.25
	Correct syntax of conditional statement/condition	0.75
	Correct syntax of else statement	0.50
	Total	2.00
Nonsyntax items		
	Declaration of variables	0.75
	Use of input statement	0.75
	Use of output statement	0.75
	Use of conditional statement/formation of the appropriate condition	1.75
	Use of else statement	1.00

(continued)

Table B3. Continued.

Elements and properties of Program 3	Points
Correct order of commands	1.00
Program execution	1.00
Correct program execution	1.00
Total	8.00

Acknowledgments

Many thanks to Dr. Aggeliki Pragiati for her overall help, support, and ideas with respect to this study. Special thanks to Vera Gerasimatou, Konstantina Plessa and Dr. Alexis Lazanas for facilitating and their overall help regarding the conduction of the empirical study.

Declaration of Conflicting Interests

The authors declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding

The authors received no financial support for the research, authorship, and/or publication of this article.

Notes

1. In the case of biologically primary information, these limitations are not so important, because the human mind was specifically evolved with the ability of processing it.
2. Although an English version has been created for a wider audience.

ORCID iD

Ioannis V. Vasilopoulos http://orcid.org/0000-0002-1332-9804

References

Ackermann, E. (2001). Constructivisme et Constructionisme: Quelle Différence (Piaget's Constructivism, Papert's Constructionism: What's the difference?) In *Constructivisms: Usages et Perspectives en Education* (Volume 1 et 2, pp. 85–94). Geneva, Switzerland: SRED/Cahier 8.

Barr, M., Holden, S., Phillips, D., & Greening, T. (1999). An exploration of novice programming errors in an object-oriented environment. *SIGCSE Bulletin*, *31*(4), 42–46. doi:10.1145/349522.349392

Ben-Ari, M., Bednarik, R., Ben-Bassat Levy, R., Ebel, G., Moreno, A., Myller, N., & Sutinen, E. (2011). A decade of research and development on program animation: The Jeliot experience. *Journal of Visual Languages & Computing*, *22*(5), 375–384.

Bishop-Clark, C., Courte, J., Evans, D., & Howard, E. V. (2007). A quantitative and qualitative investigation of using Alice programming to improve confidence, enjoyment and achievement among non-majors. *Journal of Educational Computing Research*, *37*(2), 193–207.

Bouvier, D., Lovellette, E., Matta, J., Alshaigy, B., Becker, B. A., Craig, M., . . . Zarb, M. (2016). Novice programmers and the problem description effect. *Proceedings of the 2016 ITICSE Working Group Reports*, Arequipa, Peru, 103–118. doi:10.1145/3024906.3024912

Brown, N. C. C., & Altadmri, A. (2017). Novice Java programming mistakes: Large-scale data vs. educator beliefs. *Transactions on Computing Education*, *17*(2), 1–21. doi:10.1145/2994154

Calloni, B. A., Bagert, D. J., & Haiduk, H. P. (1997). Iconic programming proves effective for teaching the first year programming sequence. *ACM SIGCSE Bulletin*, *29*(1), 262–266. doi:10.1145/268085.268189

Carlisle, M. C., Wilson, T. A., Humphries, J. W., & Hadfield, S. M. (2005). RAPTOR: A visual programming environment for teaching algorithmic problem solving. *ACM SIGCSE Bulletin*, *37*(1), 176–180.

Carroll, J. M., & Carrithers, C. (1984). Training wheels in a user interface. *Communications of the ACM*, *27*(8), 800–806. doi:10.1145/358198.358218

Caspersen, M. E., & Bennedsen, J. (2007). *Instructional design of a programming course: A learning theoretic approach*. Paper presented at the Third International Workshop on Computing Education Research, Atlanta, Georgia. doi:10.1145/1288580.1288595

Chih-Kai, C. (2014). Effects of using Alice and scratch in an introductory programming course for corrective instruction. *Journal of Educational Computing Research*, *51*(2), 185–204. doi:10.2190/EC.51.2.c

Chilana, P. K., Alcock, C., Dembla, S., Ho, A., Hurst, A., Armstrong, B., & J. Guo, P. (2015). *Perceptions of non-CS majors in intro programming: The rise of the conversational programmer*. Paper presented at the 2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), Atlanta, GA. doi:10.1109/VLHCC.2015.7357224

Cohen, J. (1988). *Statistical Power Analysis for the Behavioral Sciences* (2nd ed.). Hillsdale, New Jersey: Erlbaum.

Crook, C., & Sutherland, R. (2017). Technology and theories of learning. In E. Duval, M. Sharples, & R. Sutherland (Eds.), *Technology enhanced learning: Research themes* (pp. 11–27). Cham, Switzerland: Springer International Publishing.

du Boulay, B., O'Shea, T., & Monk, J. (1999). The black box inside the glass box: Presenting computing concepts to novices. *International Journal of Human-Computer Studies*, *51*(2), 265–227.

Efopoulos, V., Dagdilelis, V., Evangelidis, G., & Satratzemi, M. (2005). *WIPE: A programming environment for novices*. Paper presented at the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education, Caparica, Portugal.

Field, A. (2017). *Discovering statistics using SPSS; and sex, drugs and rock'n'roll* (5th ed.). London, England: Sage.

Garner, S. (2009). A quantitative study of a software tool that supports a part-complete solution method on learning outcomes. *Journal of Information Technology Education*, *8*, 285–310.

Gaspar, A., Langevin, S., & Boyer, N. (2008). Redundancy and syntax-late approaches in introductory programming courses. *Journal of Computing Sciences in Colleges*, *24*(2), 204–212.

Gosling, J., & McGilton, H. (1996). *The Java language environment*. Retrieved from http://www.oracle.com/technetwork/java/langenv-140151.html

Grandell, L., Peltomäki, M., & Salakoski, T. (2005). *High-school programming—A beyond-syntax analysis of novice programmers' difficulties*. Paper presented at the Fifth Koli Calling International Conference on Computer Science Education, Koli, Finland.

Green, T. R. G. (1990). The nature of programming. In J.-M. Hoc, T. R. G. Green, R. Samurçay, & D. J. Gilmore (Eds.), *Psychology of programming* (pp. 23–44). Retrieved from http://www.cl.cam.ac.uk/teaching/1011/R201/

Green, T. R. G., & Petre, M. (1996). Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Languages & Computing*, *7*(2), 131–174. doi:10.1006/jvlc.1996.0009

Hendrix, T. D., Cross, J. H., II., & Larry, A. B. (2004). An extensible framework for providing dynamic data structure visualizations in a lightweight IDE. *SIGCSE Bulletin, 36*(1), 387–391. doi:10.1145/1028174.971433

Hewett, T. T. (2005). Informing the design of computer-based environments to support creativity. *International Journal of Human-Computer Studies*, *63*(4–5), 383–409. doi:10.1016/j.ijhcs.2005.04.004

Hromkovič, J., Serafini, G., & Staub, J. (2017). *XLogoOnline: A single-page, browser-based programming environment for schools aiming at reducing cognitive load on pupils*. Paper presented at the Informatics in Schools: Focus on Learning Programming, Cham, Switzerland. doi:10.1007/978-3-319-71483-7_18

Hsu, J. (2003). User interfaces and markup language programming: The effects of interaction mode on user performance and satisfaction. In G. Steven (Ed.), *Computing information technology: The human side* (pp. 78–109). Hershey, PA: IGI Global.

Hundhausen, C. D., & Brown, J. L. (2007). What you see is what you code: A "live" algorithm development and visualization environment for novice learners. *Journal of Visual Languages & Computing*, *18*(1), 22–47. doi:10.1016/j.jvlc.2006.03.002

Hundhausen, C. D., Farley, S. F., & Brown, J. L. (2009). Can direct manipulation lower the barriers to computer programming and promote transfer of training? An experimental study. *ACM Transactions on Computer-Human Interaction*, *16*(3), 1–40. doi:10.1145/1592440.1592442

Kahler, S. E. (2002). *A comparison of knowledge acquisition methods for the elicitation of procedural mental models* (PhD thesis). North Carolina State University, Raleigh. Retrieved from https://repository.lib.ncsu.edu/handle/1840.16/4900

Kay, D., & Kibble, J. (2016). Learning theories 101: Application to everyday teaching and scholarship. *Advances in Physiology Education*, *40*(1), 17–25. doi:10.1152/advan.00132.2015

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.

Kölling, M. (2010). The Greenfoot programming environment. *ACM Transactions on Computing Education*, *10*(4), 1–21. doi:10.1145/1868358.1868361

Kordaki, M. (2010). A drawing and multi-representational computer environment for beginners' learning of programming using C: Design and pilot formative evaluation. *Computers & Education*, *54*(1), 69–87.

Koulouri, T., Lauria, S., & Macredie, R. D. (2014). Teaching introductory programming: A quantitative evaluation of different approaches. *Transactions on Computing Education*, *14*(4), 1–28. doi:10.1145/2662412

Lau, W. W. F., & Yuen, A. H. K. (2011). Modelling programming performance: Beyond the influence of learner characteristics. *Computers & Education*, *57*(1), 1202–1213.

LCSI. (2008). *LCSI - Solutions - MicroWorlds Pro*. Retrieved from http://www.microworlds.com/solutions/mwpro.html

Lee, M. J., & Ko, A. J. (2011). *Personifying programming tool feedback improves novice programmers' learning*. Paper presented at the Seventh International Workshop on Computing Education Research, Providence, Rhode Island. doi:10.1145/2016911.2016934

Linn, M. C. (1985). The cognitive consequences of programming instruction in classrooms. *Educational Researcher*, *14*(5), 14–29.

Lye, S. Y., & Koh, J. H. L. (2014). Review on teaching and learning of computational thinking through programming: What is next for K-12? *Computers in Human Behavior*, *41*, 51–61. doi:10.1016/j.chb.2014.09.012

Mannila, L., & de Raadt, M. (2006). *An objective comparison of languages for teaching introductory programming*. Paper presented at the Sixth Baltic Sea Conference on Computing Education Research: Koli Calling 2006, Uppsala, Sweden. doi:10.1145/1315803.1315811

Mason, R., Cooper, G., Simon, B., & Wilks, B. (2015). Using cognitive load theory to select an environment for teaching mobile apps development. In D. D. S. K. Falkner (Ed.), *Proceedings of the 17th Australasian Computing Education Conference in Research and Practice in Information Technology (CRPIT) Series* (Vol. 160, pp. 47–56). Sydney, Australia: Australian Computer Society.

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B.-D . . . Wilusz, T. (2001). *A multi-national, multi-institutional study of assessment of programming skills of first-year CS students*. Paper presented at the Conference on Innovation and Technology in Computer Science Education, Canterbury, UK. doi:10.1145/572133.572137

McIver, L. (2000). *The effect of programming language on error rates of novice programmers*. Paper presented at the 12th Annual Workshop of Psychology of Programmers Interest Group (PPIG), Corigliano.

McIver, L., & Conway, D. (1996). *Seven deadly sins of introductory programming language design*. Paper presented at the International Conference on Software Engineering: Education and Practice, Dunedin, New Zealand. doi:10.1109/SEEP.1996.534015

Milne, I., & Rowe, G. (2002). Difficulties in learning and teaching programming—Views of students and tutors. *Education and Information Technologies*, *7*(1), 55–66.

Moons, J., & De Backer, C. (2013). The design and pilot evaluation of an interactive learning environment for introductory programming influenced by cognitive load theory and constructivism. *Computers & Education*, *60*(1), 368–384. doi:10.1016/j.compedu.2012.08.009

Morrison, B. (2017). Dual modality code explanations for novices: Unexpected results. *Proceedings of the 2017 ACM Conference on International Computing Education Research* (pp. 226–235). New York, NY: ACM.

Morrison, B. B. (2015). Computer science is different!: Educational psychology experiments do not reliably replicate in programming domain. *Proceedings of the Eleventh Annual International Conference on International Computing Education Research* (pp. 267–268). New York, NY: ACM.

Morrison, B. B., Decker, A., & Margulieux, L. E. (2016). Learning loops: A replication study illuminates impact of HS courses. *Proceedings of the 2016 ACM Conference on International Computing Education Research* (pp. 221–230). New York, NY: ACM.

Myers, B. A. (1990). Taxonomies of visual programming and program visualization. *Journal of Visual Languages & Computing*, *1*(1), 97–123. doi:10.1016/S1045-926X(05)80036-9

Nikolaidhs, S. (2008). SpiNet - ΓλωσσοΜάθεια [SpiNet - Glossomatheia]. Retrieved from http://www.spinet.gr/glossomatheia/

Paas, F., & Sweller, J. (2012). An evolutionary upgrade of cognitive load theory: Using the human motor system and collaboration to support the learning of complex cognitive tasks. *Educational Psychology Review*, *24*(1), 27–45.

Paas, F., & van Merriënboer, J. J. G. (1994). Variability of worked examples and transfer of geometrical problem-solving skills: A cognitive-load approach. *Journal of Educational Psychology*, *86*(1), 122–133.

Pane, J. F., & Myers, B. A. (1996). *Usability issues in the design of novice programming systems* (School of Computer Science Technical Report CMU-CS-96-132). Pittsburgh, PA: Carnegie Mellon University.

Pane, J. F., Myers, B. A., & Ratanamahatana, C. A. (2001). Studying the language and structure in non-programmers' solutions to programming problems. *International Journal of Human-Computer Studies*, *54*(2), 237–264. doi:10.1006/ijhc.2000.0410

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

Pedhazur, E. J., & Pedhazur Schmelkin, L. (1991). *Measurement, design, and analysis: An integrated approach*. Hillside, NJ: Erlbaum.

Petre, M., Blackwell, A. F., & Green, T. R. G. (1998). Cognitive questions in software visualisation. In J. Stasko, J. Domingue, M. Brown, & B. A. Price (Eds.), *Software visualization: Programming as a multi-media experience* (pp. 453–480). Cambridge, MA: MIT Press.

Pillay, N., & Jugoo, V. R. (2005). An investigation into student characteristics affecting novice programming performance. *SIGCSE Bulletin*, *37*(4), 107–110. doi:10.1145/1113847.1113888

Rajala, T., Laakso, M.-J., Kaila, E., & Salakoski, T. (2008). Effectiveness of program visualization: A case study with the ViLLE tool. *Journal of Information Technology Education*, *7*, 15–32.

Raskin, J. (2005). Comments are more important than code. *Queue*, *3*(2), 64–65. doi:10.1145/1053331.1053354

Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., . . . Kafai, Y. (2009). Scratch: Programming for all. *Communications of the ACM*, *52*(11), 60–67. doi:10.1145/1592761.1592779

Rist, R. (1996). Teaching Eiffel as a first language. *Journal of Object-Oriented Programming*, *9*(1), 30–41.

Robins, A., Rountree, J., & Rountree, N. (2003). Learning and teaching programming: A review and discussion. *Computer Science Education*, *13*(2), 137–172.

Roussel, S., Joulia, D., Tricot, A., & Sweller, J. (2017). Learning subject content through a foreign language should not ignore human cognitive architecture: A cognitive load theory approach. *Learning and Instruction*, *52*, 69–79. doi:10.1016/j.learninstruc.2017.04.007

Scarr, J., Cockburn, A., Gutwin, C., & Quinn, P. (2011). Dips and ceilings: Understanding and supporting transitions to expertise in user interfaces. *Proceedings of the CHI 2011 Conference on Human Factors in Computing Systems* (pp. 2741–2750). New York, NY: ACM.

Scott, A. (2010). *Using flowcharts, code and animation for improved comprehension and ability in novice programming* (PhD thesis). University of Glamorgan, South Wales. Retrieved from https://pure.southwales.ac.uk/en/studentthesis/using-flowcharts-code-and-animation-for-improved-comprehension-and-ability-in-novice-programming(9324210b-c420-4752-9df0-4e6c70c43f2b).html

Shaffer, D., Doube, W., & Tuovinen, J. (2003). *Applying cognitive load theory to computer science education*. Paper presented at the 15th Workshop of the Psychology of Programming Interest Group (PPIG), Keele, UK.

Sjøberg, S. (2007). Constructivism and learning. In E. Baker, B. McGaw, & P. Peterson (Eds.), *International encyclopaedia of education* (3rd ed., pp. 485–490). Retrieved from http://folk.uio.no/sveinsj/Constructivism_and_learning_Sjoberg.pdf

Sorva, J., Karavirta, V., & Malmi, L. (2013). A review of generic program visualization systems for introductory programming education. *ACM Transactions on Computing Education*, *13*(4), 1–64. doi:10.1145/2490822

Stefik, A., & Hanenberg, S. (2014). The programming language wars: Questions and responsibilities for the programming language community. *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (pp. 283–299). New York, NY: ACM.

Stefik, A., Hundhausen, C., & Patterson, R. (2011). An empirical investigation into the design of auditory cues to enhance computer program comprehension. *International Journal of Human-Computer Studies, 69*(12), 820–838.

Stefik, A., & Siebert, S. (2013). An empirical investigation into programming language syntax. *Transactions on Computing Education*, *13*(4), 1–40. doi:10.1145/2534973

Sweller, J. (1988). Cognitive load during problem solving: Effects on learning. *Cognitive Science*, *12*(2), 257–285.

Sweller, J. (2010). Element interactivity and intrinsic, extraneous, and germane cognitive load. *Educational Psychology Review*, *22*(2), 123–138.

Sweller, J., Ayres, P., & Kalyuga, S. (2011). Intrinsic and extraneous cognitive load. In J. Sweller, P. Ayres, & S. Kalyuga (Eds.), *Cognitive load theory* (pp. 57–69). New York, NY: Springer.

Sweller, J., van Merriënboer, J. J. G., & Paas, F. (1998). Cognitive architecture and instructional design. *Educational Psychology Review*, *10*(3), 251–296.

Tudoreanu, M. E., & Kraemer, E. (2008). Balanced cognitive load significantly improves the effectiveness of algorithm animation as a problem-solving tool. *Journal of Visual Languages & Computing*, *19*(5), 598–616. doi:10.1016/j.jvlc.2008.01.001

van Merriënboer, J. J. G. (1990). Instructional strategies for teaching computer programming: Interactions with the cognitive style reflection-impulsivity. *Journal of Research on Computing in Education*, *23*(1), 45–53.

van Merriënboer, J. J. G., & Sweller, J. (2005). Cognitive load theory and complex learning: Recent developments and future directions. *Educational Psychology Review*, *17*(2), 147–178.

van Mierlo, C., Jarodzka, H., Kirschner, F., & Kirschner, P. A. (2011). Cognitive load theory and e-learning. In Z. Yan (Ed.), *Encyclopedia of cyberbehavior* (pp. 1178–1211). Hershey, PA: IGI Global.

Vasilopoulos, I. V. (2014). *The design, development and evaluation of a visual programming tool for novice programmers: Psychological and pedagogical effects of introductory programming tools on programming knowledge of Greek students* (PhD thesis). Teesside University, Teesside, UK.

Virtanen, A. T., Lahtinen, E., & Järvinen, H.-M. (2005). *VIP, a visual interpreter for learning introductory programming with C++*. Paper presented at the Koli Calling 2005 International Conference on Computer Science Education, Koli, Finland.

Vogts, D., Calitz, A., & Greyling, J. (2008). *Comparison of the effects of professional and pedagogical program development environments on novice programmers*. Paper presented at the 2008 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on IT research in Developing Countries: Riding the Wave of Technology, Wilderness, South Africa.

Vrachnos, E., & Jimoyiannis, A. (2008). *DAVE: A dynamic algorithm visualization environment for novice learners*. Paper presented at the Eighth IEEE International Conference on Advanced Learning Technologies, Santander, Cantabria, Spain.

Wiedenbeck, S., Ramalingam, V., Sarasamma, S., & Corritore, C. (1999). A comparison of the comprehension of object-oriented and procedural programs by novice programmers. *Interacting with Computers*, *11*(3), 255–282. doi:10.1016/S0953-5438(98)00029-0

Williams, J. S. (2014). *A computer learning environment for novice java programmers that supports cognitive load reducing adaptations and dynamic visualizations of computer memory* (Doctor of Philosophy). University of Wisconsin-Milwaukee. Retrieved from https://dc.uwm.edu/cgi/viewcontent.cgi?article=1579&context=etd

Winslow, L. E. (1996). Programming pedagogy—A psychological overview. *ACM SIGCSE Bulletin*, *28*(3), 17–22.

Ziegler, U., & Crews, T. (1999). An integrated program development tool for teaching and learning how to program. *SIGCSE Bulletin*, *31*(1), 276–280. doi:10.1145/384266.299786

Author Biographies

Ioannis V. Vasilopoulos, PhD, is a computer science teacher with research interests in educational computer programing software, psychology of programming, and cognitive psychology.

Paul van Schaik, PhD, is a professor of psychology at Teesside University. His research interests include human–computer interaction, judgment and decision-making, behavioral computer security, and behavioral information privacy.