Using SOLO taxonomy to explore students' mental models of the programming variable and the assignment statement

Athanassios Jimoyiannis
ajimoyia@uop.gr

Department of Social and Educational Policy, University of Peloponnese, Greece

Abstract. Introductory programming seems far from being successful at both university and high school levels. Research data already published offer significant knowledge regarding university students' deficiencies in computer programming and the alternative representations they built about abstract programming constructs. However, secondary education students' learning and development in computer programming has not been extensively studied. This paper reports on the use of the SOLO taxonomy to explore secondary education students' representations of the concept of programming variable and the assignment statement. Data was collected in the form of students' written responses to programming tasks related to short code programs. The responses were mapped to the different levels of the SOLO taxonomy. The results showed that approximately more than one half of the students in the sample tended to manifest prestructural, unistructural and multistructural responses to the research tasks. In addition, the findings provide evidence that students' thinking and application patterns are prevalently based on mathematical-like mental models about the concepts of programming variable and the assignment statement. The paper concludes with suggestions for instructional design and practice to help students' building coherent and viable mental models of the programming variable and the assignment statement.

Keywords: Introductory programming, SOLO taxonomy, mental models**,** programming variable, assignment statement

Introduction

Understanding how students learn to program has been a central topic in computer science education for decades. Early research was focused on the psychology of programming and studied the differences between novices and experts to solve programming problems (Hoc et al., 1990; Rist, 1989; Soloway, 1986). Independent research studies into students' programming ability, in various countries and educational contexts, have shown that university students have similar difficulties in writing, tracing and designing programs (McCracken et al., 2001; Chalk et al., 2003; Lister et al., 2004; Eckerdal et al., 2006; de Raadt, 2007). From a cognitive perspective, existing research has given interesting findings regarding students' difficulties, misconceptions or alternative conceptions when using abstract programming concepts and constructs (e.g. variables, control structures, loops, arrays, recursion), students' inadequate planning skills and deficits s in programming for problem solving, the common bugs/errors novices exhibit while programming and testing their code etc. (Ebrahimi, 1994; Lane & VanLehn, 2005; Corritore & Wiedenbeck, 1991; Postner & Stevens, 2005; Rogalski & Samurcay, 1990; Sleeman et al., 1986; Spohrer & Soloway, 1989).

Investigating students' development from concrete representations of programming concepts to abstract mental models is one of the principal themes in computer science education research (Cortney et al., 2011). However, existing research experiments did not adequately reveal the many aspects of students' programming thinking. Assessing the

degree of novices' ability to see the relationships between abstract programming concepts and the constitutional parts of a program is currently an open research program internationally (Lister, Fidge & Teague, 2009; Ma et al., 2011; Sheard et al., 2008). In addition, addressing new instructional approaches and techniques that help students' learning is still an issue of interest in computer science education.

Towards this direction, the Structure of the Observed Learning Outcome (SOLO) taxonomy proposed by Biggs & Collis (1982) was emerged in the recent years as a promising educational taxonomy to study how novice programmers manifest their understanding of programming constructs (Lister et al., 2006; 2009a; Thomson, 2007; Whalley et al., 2006). Research findings have demonstrated that SOLO is a relevant and efficient framework for studying students' representations of programming concepts and studying their development in programming (Lister et al., 2006; 2009; Sheard et al., 2008).

Despite that university students' learning and development in computer programming has been extensively studied, the relation of secondary education students and programming instruction, in both the Greek educational context and internationally, is an under-researched topic. Using the SOLO taxonomy, this study examines secondary education (K-12) students' programming thinking and their representations of the concepts of programming variable and the assignment statement. Data was collected in the form of students' written responses using a questionnaire testing their ability to predict the outcome of executing short code programming tasks. The paper provides evidence that the mathematical mental models of the concepts of programming variable and the assignment statement are prevalent between the students in the sample.

The article is structured as follows: The first section presents a literature review addressing both theoretical foundations and students' difficulties in programming and, specifically, in using the programming variable and the assignment statement to solve typical programming problems. Following, the methodology of the research is presented. Finally, the research data analysed using the SOLO schema and the findings are presented. Conclusions are drawn for educational practice in introductory programming courses and further research in the area.

Literature review

Experts versus novice programmers

Previous investigations found that many students, not only at secondary education but also at university level, confront critical difficulties in using abstract concepts, they hold non efficient mental models of basic programming concepts and perform more poorly than expected after experiencing programming courses. A study conducted in four universities (McCracken, 2001) concluded that many computer science students, at the end of their introductory courses, do not know how to program and lack the skills needed for getting a program ready to run. Another research in seven countries by Lister et al. (2004) reiterated similar findings showing that many students have a fragile grasp of both basic programming principles and the ability to systematically carry out routine programming tasks, such as code tracing. In addition, Simon et al. (2006) have looked for predictors of students' programming success and found only very weak indicators.

There is a considerable amount of evidence that novices have severe problems in understanding the basic concepts of programming. Most investigations in the decade of 80's have focused on the psychology of programming and have given considerable information about students' alternative conceptions of programming structures (Soloway & Spohrer,

1989; Hoc et al., 1990) and the cognitive difficulties they encounter when they solve programming problems (Samurçay, 1989; Soloway & Spohrer, 1989; Hoc et al., 1990). The classical book entitled "Studying the novice programmer", edited by Soloway & Spohrer (1989), collected various papers outlining the deficits of students in understanding basic programming constructs (variables, loops, arrays, recursion), revealing the shortcomings in code planning and testing, and showing how prior knowledge can be a source of errors for novices.

It is a common conclusion that students have faulty or fragile mental models concerning programming constructs, objects, attributes, and methods while they exhibit poor performance in using elementary problem-solving strategies (Perkins et al., 1989; Eckerdal & Thune, 2005; Eckerdal et al., 2006; Holland et al., 1997; Garner, Haden & Robins, 2005). The majority of students face at serious difficulties and lack the skills necessary to function abstractively, to consolidate a program as a single entity, to comprehend its main parts and the relations among them, to compose new algorithms and to effectively adapt statements or procedures using their previous programming knowledge (Robins et al., 2003).

The differences between novices and experts in computer programming have been studied extensively and tend to confirm that novices do not have many of the abilities of experts (Brooks, 1990; Gilmore, 1990; Putman et al., 1989; Rist, 1991; Wiedenbeck et al., 1993; Winslow, 1996). Research findings indicate that expert programmers form abstract representations based upon the purpose of the code whereas novices form concrete representations based on how the code functions. On the other hand, novice programmers are "very local and concrete in their comprehension of programs" (Wiedenbeck & Ramalingam, 1999). The emergence of the expert programmer from the novice is a process that involves the formation of multiple mental models, deep and interlinked knowledge hierarchies, and abstract, generalized problem-solving patterns (Winslow, 1996). It is a commonplace that traditional instruction of introductory programming, at both universities and secondary schools, seem to be less successful than needed. McGettrick et al. (2005) note that educators cited failure in introductory programming courses and/or disenchantment with programming as major factors underlying the poor student retention in computing degree programmes.

One possible reason for this is that novices hold 'non-viable' mental models of key programming concepts which then cause misconceptions and difficulties (Ma et al. 2009). Given a problem, the expert programmer can retrieve an appropriate solution schema. In contrast, novice programmers are often restricted to a language or syntax-oriented organization of their programming knowledge. This kind of knowledge does not allow a new problem to be matched with a previously learned solution (Soloway, 1986; Rist, 1989). As a result, novices have difficulties in assembling algorithms. In Table 1, we summarize the most critical issues that shape the differences-alternative approaches in computer programming between expert and novice programmers.

Secondary education students' barriers in programming

This section outlines an educational framework regarding computer programming in secondary education. This framework is addressed mainly to the Greek context but many of the ideas could be of interest and transferable to other educational contexts. Unlike to the fields of science and mathematics education, which have developed a coherent framework of knowledge about how secondary education students understand and learn these subjects, computer science educators in Greece are just beginning to consider the complex issues involved in learning to program.

Table 1. Major differences in programming between expert and novice programmers

Expert programmers	Students-novice programmers
Rely on plans or schemata to understand programs	Tend to focus on the syntactic details of the code and execute code line-by-line
Integrate the parts of a program into a coherent structure	May understand the parts of a program but struggle to organize those parts into a coherent structure
Organize their programming knowledge in sophisticated, flexible and viable mental models	Lack detailed and viable mental models Are limited to surface and superficially organised programming knowledge
Organize their representations of code segments into larger conceptual-programming structures according to the function performed by the code	Form concrete (local) representations based on how the code functions
Focus on algorithmic approaches (algorithms, models, schemata) rather than on commands and the syntactic details of the particular programming language	Lack algorithmic thinking and fail to apply relevant knowledge to solve problems Approach programming as line-by-line code execution rather than using meaningful program structures
Have a wide repertory of algorithms applicable in solving new problems	Do not easily develop patterns (algorithms) that allow them to match a problem with a previously learned solution

Current research data concerning students' misconceptions and mental models for programming concepts indicate that teaching computer programming to secondary students includes many more things than the syntactic details and the semantics of the specific programming language used. In computer programming we are thinking about algorithms and data in ways that are very different from those in other cognitive activities or areas (e.g. mathematics or physics) and, most of all, in our everyday life. Students' pre-existing knowledge and experiences, coming from other related subjects play an important role in their efforts to develop algorithmic thinking (Bonar & Soloway, 1985). However, in most of the cases, this is not enough. For example, in programming students need to manipulate many abstract entities that have no or little relation to their pre-existing knowledge and everyday experience (e.g. logical data type, nested control structures, looping constructs, initialization of variables, counters, arrays and pointers, recursion, etc.). These entities concern, on the one hand, the programming language used and, on the other, how statements should be used to produce meaningful combinations in order to construct a complete program solving a problem.

It is reasonable, therefore, that students have many difficulties when they try to express solutions which do not come spontaneously or have not previously been familiarized with, e.g. as the natural consequence of the knowledge transferred to programming from other cognitive subjects. For example, let us compare a typical Pascal or C program calculating the summary of a series of numbers to the similar mathematical (by hand) procedure or to the SUM function used in a spreadsheet. The instructional experience of the author indicates that this particular mismatch in the way of thinking about solutions, e.g. how to express a solution in a programming language, is a critical barrier for most secondary students.

In addition, students have no ability to negotiate the syntactic rules and the semantics of the particular programming language used. In order to solve problems effectively, they need to adapt their solutions to the syntactic and structural strictness of the programming environment. This is inherent to computer programming and it is not easily achieved by the majority of students.

Another source of difficulties is related to the representations of the students about the concept of the program itself, which is a mechanism able to solve a problem when executed correctly. The role of the machine (computer) is not easily accessible by the students, at least in the introductory lessons. The computer and the programming environment have a two-fold role: they constitute a mechanism which can be used for both developing and executing other mechanisms (programs). Du Boulay (1986) introduced the term of *notional machine* to describe the role of the computer in programming and the related perceptions of novices. Jimoyiannis & Komis (2004) have shown that secondary education students lack efficient representations of the machine and its internal operation during the execution of a program. Students' faulty models of the notional machine constitute a factor inducing significant barriers to their programming thinking.

The last issue concerns the typical programming environments and languages used in instruction. They have been constructed for developing software applications rather than for educational purposes. Consequently, programming environments are adapted to a framework of knowledge and skills achieved by experienced programmers. This is a main source of difficulties and obstacles for students as novices in programming. In the last decade, there is a growing interest about using alternative instructional environments in introductory programming courses; pseudocode and/or micro-languages (e.g. Logo, Karel, BlueJ), educational programming environments (e.g. MicroWolds Pro, RoboLab, Scratch, BYOB), and algorithmic simulation environments, like Jeliot (Moreno et al., 2004), JHAVE (Naps et al., 2003), Alvis (Hundhausen & Brown, 2007) etc.

The concepts of variable and the assignment statement

Teaching programming in secondary education constitutes a very interesting task with specific characteristics and differences comparing with the other subjects in K-12 Curriculum. Algorithms are arguably the cornerstone of computer science and programming. The effective use of variables is fundamental to computer programming and the design of algorithms. Students' efficient and viable models about variables is an essential prerequisite to built other abstract constructs, e.g. counters, arrays, loops etc. However, the use of variables in programming and especially their role in the program is a tacit implicit knowledge that cannot be presented explicitly to students (Sajaniemi, 2002; Ma et al., 2011; Sheard et al., 2008).

Early research findings have shown that secondary education students have various difficulties to manipulate variables when they try to solve simple programming problems (Samurçay, 1989; Soloway & Spohrer, 1989; Green 1990; Palumbo & Reed, 1991). Samurçay (1989) investigated the different ways variables are related to assignment statements and described the mental difficulties of novice programmers about dynamic modifications of variables, like updating and initialisation. In addition, Du Boulay (1986) identified misconceptions about variables, based upon the analogies used in class. Important issues were revealed, like the role of initialisation, linking variables by assigning, misunderstanding of temporal scope of variables etc. Perkins et al. (1989) reported other form of misconceptions related to the names of variables.

In addition, Dehnadi and Bornat (2009) reported that students hold eleven different mental models for the assignment statements. They argued that they had found a relationship between the consistency of the mental models employed and students' performance in programming examinations. A recent survey on university students, enrolled in an introductory computing course (Corney et al., 2011), reported that, in the third course week, almost half of the students could not respond efficiently to a code consisting of just three assignment statements, which swapped the values in two variables.

Although students face at variables during early introductory programming courses, the construction of efficient mental models appears to be a difficult task. Data processing and storage in the form of variables, as perceived by the students through the use of symbols, constitutes a main source of difficulties. The concept of programming variable is usually built on students' pre-existing mathematical knowledge. In most cases, the first problems that students are asked to solve in introductory lessons have a typical mathematical nature. Moreover, the type of names used in the first programs are similar to those in mathematical problems (e.g. x, y, z, a, b). On the other hand, the familiar (by hand) calculation processes, well-known from mathematics, are used in introductory lessons with the objective to support building of the concepts of variable and the assignment statement in students' minds. These processes constitute a critical cognitive obstacle that students need also to overcome.

It appears that the common perception of students about programming variable and the assignment statement concepts is restricted to the mathematical representation, even after many lessons in programming (Jimoyiannis & Komis, 2000). However, in mathematics a variable has a static meaning and presents a mathematical-functional relation. This view is not adequate for the students to comprehend the functional and dynamic meaning of programming variable (Rogalski & Vergnaud, 1987). A programming variable is an abstraction of a memory cell, i.e. an associated memory location inside the computer having a constant capacity and containing a value of a data type predefined via a declaration statement. During program execution, the value of a variable can be dynamically modified as it is manipulated by assignment or input (read) statements.

The majority of students have not comprehended the dynamic interrelation between the variable and the assignment concepts, e.g. through the assignment statement data are registered upon the pre-existing value of a variable, which is lost afterwards. Many students think that "variables can store more than one value at a time" or that "a variable can 'remember' the history of previous assignments". In other words, they have a representation of stack-type about the programming variable and, consequently, they believe that they can recover all those previous values (Jimoyiannis, 2000).

The assignment statement is a command for assigning a value to a variable. The assignment statement has also a mathematical sense, which emanates from the name of the variables involved and the symbol used for the assignment operator (=, :=, ←), which is often confused with the mathematical symbol of equality. However, there is a main difference in the deep meaning of the assignment operation and the symbols related to the variables included. For example, the simple assignment statement B ← A can be misinterpreted by students as after the assignment statement "variables A and B are exchanged" or "the variable A no longer contains any value".

Let us consider another classical example of misconceptions for many secondary students: the statement x ← x + 5. The majority of the students are not able to comprehend that x does not concern the same entity in both sides. In the left part, x is related to the memory location; in the right part, x represents the current value of the variable x.

Swapping is another common example of students' misconceptions in most introductory classes. A great number of students struggle in the early lessons with the code for swapping the value of two variables.

Finally, the nature of the data involved in a problem with variables introduces additional difficulties. For example, students can more easily manipulate variables representing numerical data (integer or real), possibly because they are related to cognitive forms that they are familiar with. On the other hand, the use of string or logical variables and,

moreover, complex structures, like arrays, demands new form of representations that students cannot easily construct (Jimoyiannis, 2000).

The SOLO taxonomy

The Structure of the Observed Learning Outcome (SOLO) taxonomy was proposed by Biggs & Collis (1982) as a general educational taxonomy. It is content independent and thus it can be used as a generic measure of understanding across different disciplines. SOLO is a developmental schema of classifying learning outcomes in terms of their complexity, thus enabling instructors to assess students' work in terms of its quality not of how many responses in a particular subject task or activity are correct (Chan et al., 2002).

SOLO taxonomy provides criteria that identify the levels of increasing complexity of students' performance for understanding when mastering new learning (Biggs, 1999). SOLO can be used not only in assessment but also in designing the curriculum in terms of the learning outcomes intended. It includes five levels revealing the structure complexity of students' knowledge as they learn. The lower levels focus on quantity (the amount the learner knows) while the higher levels focus on the integration, the development of relationships between the details and other concepts outside the learning domain (the integration of the details into a structural pattern).

As a general educational taxonomy, this schema is assumed to apply to any subject area. Existing research provides supportive evidence of the potential of SOLO taxonomy in the evaluation of students' learning in various subjects, like mathematics, science, technology etc. (Chick, 1998; Hazel, Prosser & Trigwell, 2002; Padiotis & Mikropoulos, 2010), learning environments and educational levels (Chen & Zimitat, 2004; Chan et al., 2002).

Lister et al. (2006) first suggested the use of SOLO taxonomy to classify students' responses to computer programming problems not so much according to their correctness as according to the level of integration that they demonstrate. The idea is that beyond the actually correct, a more integrated answer is a convincing demonstration that the student has understood the programming code. In this context, SOLO taxonomy describes five levels of student understanding when solving programming problems. Following, we present a similar interpretation of how SOLO taxonomy applies when novices manifest their understanding of short code problems.

Prestructural: This is the least sophisticated type of response a student may give to a programming task. A prestructural response manifests either a significant misconception of programming or a preconception that is irrelevant to programming. The student lacks knowledge of programming constructs and approaches the task under study in a non appropriate or unrelated way.

Unistructural: This is a response where the student manifests a correct grasp of some but not all aspects of the programming problem. The student has a partial understanding and one or few aspects are picked up and used effectually. In many cases, students make what is called an "educated guess" (Lister et al., 2004). For example, the student describes the functioning of a part (one or two lines) of the code.

Multistructural: The student focuses on several relevant aspects but does not manifest an awareness of the relationships between them or the whole. According to Lister et al. (2006), the student fails "to see the forest for the trees". For example, a student may provide a line-by-line description of the code or execute the code by hand and arrive at a final value for a particular variable. However, he is not able to see the code as a single coherent construct.

Relational: This level corresponds to what is normally meant by adequate understanding of the topic under study. The student makes sense of the various aspects of the topic, integrates the parts of the problem into a coherent code structure, and uses this structure to solve the problem. According to Lister et al. (2006), "the student sees the forest". A relational response is the most sophisticated type of response a student may give and may be either correct or incorrect. For example, a student is able to describe the function performed by a particular code segment without hand executing. The student may infer that the code counts the number of elements in an array which are greater than a particular value.

Extended Abstract: In this highest SOLO level, the student response goes beyond the particular problem to be solved, and links the problem to a broader context. The student is able to extrapolate, to develop higher order principles and extend the topic to wider application areas. For example, a possible extended abstract response may be a comment that the code will only work for arrays that are sorted (Lister et al., 2006). While this is a very interesting level to study, the tasks designed for the present study do not aimed at promoting students' performance in the extended abstract level.

Research design and methodology

Research context

Computer Science was introduced in Greek upper secondary education (Lyceum) on 1998, in the framework a new National Curriculum for upper secondary education (CF, 1998). Two elective courses, related to introductory computer science and information and communication technologies respectively, were established in the first and the second years of Lyceum (K-10, K-11). In the third year of Lyceum (K-12), the students in the technological direction need to attend an obligatory course, for 2 hours per week, under the title "Development of Applications in Programming Environments" (DAPE). It is an introductory course to procedural programming and algorithmic problem solving. The subject content is included in the national university entrance examinations.

According to the Computer Science Curriculum (CF, 1998), the main objectives of the instruction of programming in upper secondary education were

- to develop students' analytical and synthetic thinking
- to help students acquiring methodological and algorithmic thinking skills
- to enhance students' abilities to solve problems using programming environments
- to develop students' creativity and imagination.

The existing National Curriculum suggests the use of a pseudocode environment (named LANGUAGE) for the instruction of the basic algorithmic and programming structures (stepwise programming, variables, control and loop structures, arrays, procedures and functions). The instruction combines both lectures and laboratory activities in which students explore problems, and develop and analyze their solutions. During the lab sessions, the students have the opportunity to use pseudocode and/or programming environments (e.g. Pascal) for practice and solving simple programming problems.

The DAPE course was offered to K-12 students for the first time on 2000. The study presented here is of particular interest considering that computer programming has a short history in Greek secondary education.

Objectives

Most of the previous studies in the area were restricted to a descriptive survey of students' ideas and misconceptions about programming constructs. Descriptive analysis of the students' responses shows only their different approaches to the various tasks. In this particular analysis we have used SOLO taxonomy because it offers a thorough qualitative insight into students' understanding of programming constructs and how they use them to solve simple programming problems.

The survey was designed to provide information to better understand the factors that may influence students' representations and effective use of variables and the assignment statement to solve programming problems. There were three main purposes justifying this investigation:

- To replicate and extend previous research findings concerning students' typical patterns of response and their representations of the programming variable and the assignment statement.
- To better understand how students typically approach short code programs and to identify topics where students commonly experience difficulties related to the assignment statement.
- To evaluate the applicability of a pedagogically sound theoretical framework (SOLO taxonomy) in studying students' performance and their difficulties to written code tasks.

Research questions

In accordance with the research objectives and consistent with the related literature, the following research questions were addressed in this study:

Research Question 1: What are the dominant mental models, built by secondary education students, regarding the programming variable and the assignment statement?

Research Question 2: To what extend are students' conceptions of the programming variable and the assignment statement mutually related?

Research Question 3: Can SOLO taxonomy offer an efficient framework to study and assess students' representations and mental models of the programming variable and the assignment statement?

The instrument

The research tool was an anonymous written questionnaire with six programming tasks based on code segments in LANGUAGE, the pseudocode environment proposed by the Curriculum and the textbook approved by the Greek Ministry of Education. The tasks developed specifically for the purpose of this study with the aim to test students' ability to predict the outcome of executing short code programs. The tasks represented students' alternative conceptions and representations known from the literature and the teaching experience of the author. Students were asked to give their responses including also a short justification.

Demographic information such as gender, age, years of computer experience, type of software applications used etc., were also requested.

The sample

The survey presented here was administered in 2005 in five upper secondary schools from the urban area of two Greek cities, namely Ioannina and Rhodes. The schools participated were randomly selected while the research sample was representative of the students. A total of 182 students, 99 (54.4%) boys and 83 (46.6%) girls, participated in the survey. The students were aged between 17-18 years and attended the third year of upper secondary education (K-12). They attended the DAPE course about algorithmic and introductory programming for 2 hours per week.

The majority of the students had no previous programming experience when entering this course. 27% of the students reported that they were engaged in programming (using Logo, Pascal, Basic or other environments) while attended the elective computer science course in the first year of Lyceum. The majority of the students in the sample used computer games and/or the Internet while only two students were also engaged into programming activities for their personal interests.

The great majority of the students reported access to a PC either at home (84.1%), at their friends (8.2%) and other places, like computer schools and internet cafés (2.8%). On the other hand, 4.9% of the students in the sample had no opportunity to use computers in places out of the school computer laboratory.

The procedure

The research presented took place about six weeks after the students had completed introductory programming lessons in their schools and, especially, the unit related to the programming variable and the assignment statement. No intervention took place before the survey. Prior to their participation in the research all students had received traditional instruction on these topics in the classroom and the computer laboratory.

All participants were volunteers. They were briefed on the purpose of this study and informed of their rights to not participate or withdraw from completing the questionnaire at anytime during the data collection. Participants took about 30 minutes to complete the written task questionnaire.

Researcher's role during students' responding was restricted to answering their questions in order to clarify the programming tasks under research. To ensure that all tasks included in the questionnaire were clearly understood, a trial run of the survey was carried out in one school. The trial group consisted of 23 K-12 students. Data from the trial were not included in the analysis.

Results and analysis

Figure 1 shows the distribution of students' responses to the research tasks according to the SOLO categories. If we assume that students' responses are a reasonably consistent reflection of how they reason about programming code, then the students in the sample manifested a relational response to the research tasks, approximately, at a percentage of 40-50%. However, more than one half of the students exhibited prestructural and unistructural responses to the tasks. It is apparent in Figure 1 that many of the weaker students showed a consistent approach to the six tasks. They were not able to abstract from concrete code representations to meaningful structures about the concept of programming variable and the assignment statement. The students' exhibited the weakest performance in Task 6. They were classified into the relational SOLO level at a percentage of 15.93%.

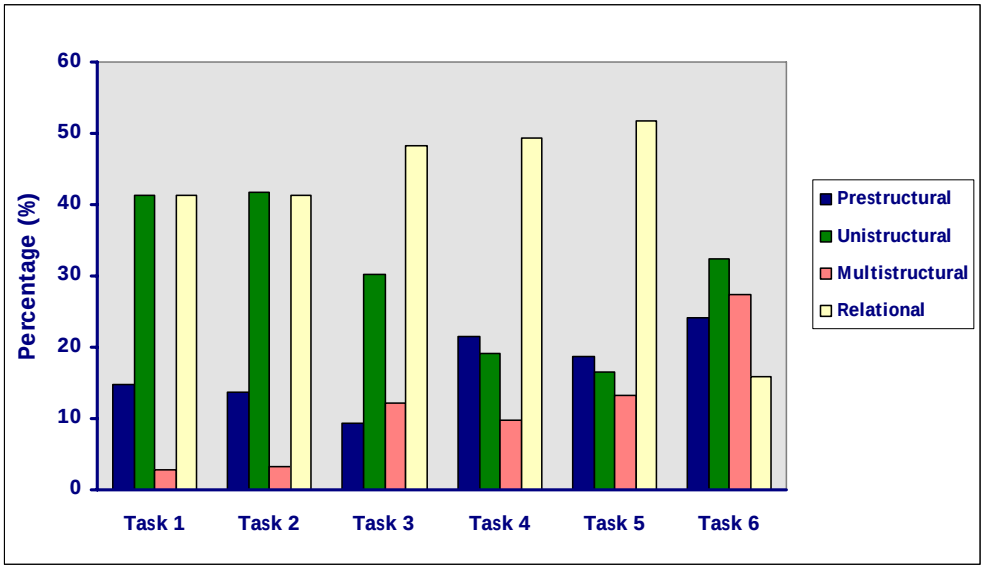

Figure 1. Distribution of students' SOLO responses to the tasks

Following we present the research findings regarding students' responses to each particular task using the SOLO schema of analysis.

Task 1

Consider the following program:

```
PROGRAM TASK1
VARIABLES
INTEGER: x, y
BEGIN
   READ y
   READ x, y
   WRITE y, x, y
END_PROGRAM TASK1
```

What do you expect to be displayed on the screen if you will repetitively input the values 3, 6, and 9? Explain your answer.

Table 2 presents students' responses to Task 1 according to SOLO taxonomy. Into the prestructural level we classified students' responses which were unrelated to programming thinking and indicated critical deficiencies. 10.99% of the students in the sample did not answer to this task. Students' justifications in this category were like

> *"There will be nothing displayed on the screen. This is a false problem because variable y cannot store two values through READ statement".*

Table 2. Students' responses to Task 1

SOLO level	Percentage% (N=182)
Prestructural	14.83
Unistructural	41.21
Multistructural	2.75
Relational	41.21
Total	100.00

We have classified as unistructural the students' responses (41.21%) which revealed a representation of the variable concept which has the form of a stack or a box. Typical responses of this SOLO level were

"There will be displayed on the screen the values 3, 6, 9".

According to the justifications given, the students above believe that, after the execution of the successive READ statements, the variable y can store two values (3 and 9). Both values were stored in variable y and can be displayed on the screen when using a WRITE statement.

2.75% of the students' responses to this task were classified into the multistructural level. They gave responses like

"There will be displayed on the screen the values 6 and 9".

These students have built an effective representation of the READ statement and were able to estimate the content of the two variables x and y. However, they exhibited deficiencies to comprehend the meaning and the outcome of the WRITE statement; they responded giving two values as the output of a WRITE statement acting on three variables (WRITE y, x, y).

41.21% of the students in the sample were classified into the relational level. They gave complete and justified answers manifesting arguments of the type

"The values that will be displayed on the screen are 9, 6 and 9. After the execution of the statement READ x, y the first input value (i.e. 3) of the variable y is lost and a new value (9) is assigned to y".

Task 2

Consider the following program:

```
PROGRAM TASK2
   VARIABLES
   INTEGER: x, y
BEGIN
   READ x
   READ x
   READ y
   WRITE y, x, x
END_PROGRAM TASK2
```

What do you expect to be displayed on the screen if you will repetitively input the values 3, 6, and 9? Explain your answer.

Task 2 is similar to Task 1. Students' reasoning and responses appeared to be consistent in both tasks. Table 3 shows the distribution of the responses to Task 2 categorized according to the SOLO taxonomy.

Into the prestructural level we classified students' responses which were unrelated to programming and indicated the same deficiencies recorded also in Task 1. The students did not answer to this task (11.54%) or they gave justifications of the type

"There will be nothing displayed on the screen. This is a false problem because variable x cannot store two values through READ statement".

We have classified as unistructural the students' responses (41.75%) which revealed a representation of the variable concept which has the form of stack or box, e.g. the variable x can store simultaneously two values (i.e. 3 and 6). Typical examples were

"There will be displayed on the screen the values 9, 3, 6".

Table 3. Students' responses to Task 2

SOLO level	Percentage% (N=182)
Prestructural	13.74
Unistructural	41.75
Multistructural	3.30
Relational	41.21
Total	100.00

"There will be displayed on the screen the values 9, 6, 3".

Similar results have also been recorded as far as the multistructural SOLO level it regards. 3.30% of the students' responses were classified in multistructural level with responses like

"There will be displayed on the screen the values 9 and 6".

Finally, 41.21% of the students in the sample were classified into the relational level. They gave complete and justified answers manifesting arguments of the type

"The values that will be displayed on the screen are 9, 6 and 6. The first input value of the variable x (i.e. 3) is lost and a new value (6) is assigned after the execution of the statement READ x".

Task 3

Consider the following program:

```
PROGRAM TASK3
   VARIABLES
   INTEGER: x,y
BEGIN
   x ← 5
   y ← x
   x ← x+5
   y ← x+5
   WRITE x,y
END_PROGRAM TASK3
```

What do you expect to be displayed on the screen when the above code finishes? Explain your answer.

Table 4 presents students responses to Task 3 categorized according to the SOLO taxonomy. We have classified in the prestructural level the students' responses which manifested conceptions irrelevant to programming. 6.59% of the students in the sample gave no answer at all. Into this category were also classified responses like

"After making the calculations I can found out the result"

"There will be displayed on the screen WRITE 10, 10".

Approximately one out of three students were classified into the second SOLO level (unistructural). It appears that the students into this category have not built a stable representation of the successive nature of the assignments and the dynamic change of value of the variables involved. Their approaches were clearly based on a *mathematical representation* of both the concept of programming variable and the assignment statement. Typical responses and justifications given were like

"Initially x=5. Consequently, the statement x←x+5 gives x=5+5=10. On the other hand, the statement y←x+5, means that y=x=10".

Table 4. Students' responses to Task 3

SOLO level	Percentage% (N=182)
Prestructural	9.34
Unistructural	30.22
Multistructural	12.09
Relational	48.35
Total	100.00

We have classified in multistructural level (12.09%) students' responses which showed correct program outcomes. However, the justification of their answers was based on a mathematical representation of the successive assignments and the program as a structure. According to their approach, the successive assignments were viewed as mathematical calculations. Indicative responses of this type were

"After doing the calculations, the final result is x=10 and y=15".

Approximately one out of two students (48.35%) gave a correct and adequately justified answer. Their responses were categorized into the relational SOLO level. Those students responded that

"The values that will be displayed on the screen are 10 and 15".

In most cases, the students used a value table showing the successive values of the variables x and y after executing the successive statements of the program.

Task 4

Consider the following code segments:

A ← 10 B ← 5 A ← B B ← A WRITE A, B	A ← 10 B ← 5 B ← A A ← B WRITE A, B

Do you expect that these code segments are equivalent or not? Explain your answer.

Table 5 presents the results of the survey concerning student's answers for Task 4. Into the prestructural level we classified students' responses indicating deficiencies and approaches unrelated to programming. A percentage of 7.69% of the students gave no answer at all. Typical examples of students' prestructural responses were

"The order of the assignments makes no difference to the values of the variables"

"The two segments are equivalent; they display the same output values, 5 and 5".

"The two segments are equivalent since they produce the same output on the screen: values 15 and 15".

The last response indicates that the students hold an 'accumulative' mental model for the assignment statement. In other words, those students perceived the assignment statement as a process in which "the value in the right is added to the content of the variable in the left part of the statement".

Into the unistructural level we have classified students' responses (27.43%) that were not based on a complete representation of the code segment. Characteristic answers of this category were

Table 5. Students' responses to Task 4

SOLO level	Percentage% (N=182)
Prestructural	21.42
Unistructural	19.23
Multistructural	9.89
Relational	49.45
Total	100.00

"The two segments are equivalent; they produce swapping of the values of the variables".

"The two segments are equivalent since they produce the same output on the screen: values 5 and 10".

"The two segments are not equivalent. The first displays on the screen the values 5 and 10 while the second displays the values 10 and 5 for the variables A and B respectively".

Into the multistructural level we have classified students' responses of the type

"The two code segments are not equivalent because the order of the assignments is different"

"The two code segments are not equivalent. After the execution of the first segment, the content of both variables is 5. In the second segment, there will be displayed on the screen the values 5 and 10 respectively".

45.14% of the students gave correct answers manifesting arguments of the type

"The two code segments are not equivalent. In the first segment, the content of both variables is 5, so there will be displayed on the screen the values 5 and 5. In the second, the content of the two variables is 5, so the values displayed on the screen are 10 and 10 respectively".

In order to justify their responses to the task, the majority of the students used a table with the successive values of the variables included in the code segment.

Task 5

Consider the following code segments:

A ← 10 B ← 5 A ← B B ← A WRITE A, B	A ← 10 B ← 5 H ← A A ← B B ← H WRITE A, B

Do you expect that these code segments are equivalent or not? Explain your answer.

Table 6 presents the results concerning students' responses to Task 5. Into the prestructural level we classified students' responses irrelative to programming thinking. Typical examples of students' prestructural responses were

"The two segments are equivalent; Variable H makes no difference".

In addition, a percentage of 6.59% of students gave no answer at all.

Into the unistructural level we have classified students' responses (27.43%) that were not based on a complete representation of the code segment. Characteristic answers of this level were

"The two segments are not equivalent. The order of the assignments to the variables is different"

Table 6. Students' responses to Task 5

SOLO level	Percentage% (N=182)
Prestructural	18.68
Unistructural	16.49
Multistructural	13.19
Relational	51.65
Total	100.00

"The two segments are not equivalent. There is an extra variable (H) in the assignments"

"The two segments are not equivalent. The first displays on the screen the values 5 and 10 while the second displays the values 10 and 5, for the variables A and B respectively".

Into the multistructural level we have classified students' responses which indicated a correct aspect of the task (the swapping code). Indicative written justifications were like the following

"The two segments are equivalent since they swap the values of the variables; they display the same output values: 5 and 10".

"The two code segments are not equivalent. The first segment will display on the screen the values 15 and 15. The second code is swapping and the values displayed on the screen will be 5 and 10 for the variables A and B respectively".

Into relational SOLO level we have classified 51.65% of the students which gave correct answers writing arguments of the type

"The two code segments are not equivalent. In the first segment, both variables' content is 5, so there will be displayed on the screen the values 5 and 5. In the second, the values displayed on the screen are 5 and 10 for the variables A and B respectively."

"The two code segments are not equivalent. In the first segment, the values 5 and 5 will be displayed on the screen. The second code produces swapping of the two variables. So, there will be displayed on the screen the values 5 and 10 for the variables A and B."

The results above reiterate recent findings reported by Corney et al. (2011) regarding university students' deficiencies about swapping of the values in two variables.

Task 6

Consider the following program:

```
PROGRAM TASK4
  VARIABLES
  INTEGER: x,y
BEGIN
  READ x, y
  x ← x + y
  y ← x – y
  x ←  x - y
  WRITE x,y
END_PROGRAM TASK4
```

What do you expect to be displayed on the screen when the above program finishes? Explain your answer.

Table 7 shows students' responses to Task 6 categorized into the SOLO levels. 24.18% of the students in the sample were classified into the prestructural level. The majority of them (21.43%) gave no answer at all. In addition, four students gave the next explanation:

Table 7. Students' responses to Task 6

SOLO level	Percentage% (N=182)
Prestructural	24.18
Unistructural	32.42
Multistructural	27.47
Relational	15.93
Total	100.00

"There will be nothing displayed on the screen. This is a false problem and cannot be solved".

Approximately one out of three students' responses was classified into the unistructural SOLO level. Those students have built a rather *mathematical representation* of both the concept of programming variable and the assignment statement. Typical responses and justifications given were like

"After executing the successive statements y←x-y and x←x-y, the two variables x, y have the same content value: x-y".

"After executing the successive statements the variable y is equal to x-y=(x-y)+y=x. Because of the statements y←x-y and x←x-y the two variables (x and y) have the same content value, i.e. x".

Into the multistructural level we have classified students' responses which indicated a correct approach and justification regarding the variable y. Those students manifested only one aspect of the task (the swapping code). Indicative written justifications were as following:

"After executing the statement y←x-y the value of the variable y is equal to x, because y=x-y= (x+y)-y=x. Following, after the execution of the statement x←x-y, the value of the variable x is x=x-y=x-x=0".

"After executing the statement y←x-y the value of the variable y is equal to x. The statement y←x-y means that y=x-y or x=2y (the students treat this relation as a mathematical equation). So, the values displayed on the screen will be 2y and x for the variables x and y respectively".

In *relational* SOLO level we have classified 15.93% of the students which gave correct answers writing arguments of the type:

"The values displayed on the screen will be y and x respectively"

"This code is swapping the two variables x and y".

The majority of the students in relational level used a value table to calculate by hand the intermediate and the final values of the variables x and y, after the successive assignments.

Discussion and conclusions

This paper reported on the investigation of K-12 students' mental models of the programming variable and the assignment statement by using SOLO taxonomy. Three major findings can be drawn from this study in relation to the research questions as follows.

The first research question examined if the students have built adequate mental models of the concepts of programming variable and the assignment statement. The majority of the students revealed difficulties in linking variables by assignment and read (input) statements. In addition, they exhibited misconceptions regarding the temporal scope of variables in a program. We have recorded three different types of faulty mental models of the concept of variable held by the students in the sample:

- The *mathematical model*, i.e. a variable is considered like a parameter involved in a calculation formula
- The *box or stack model*, i.e. a variable can store more than one values at a time
- The *static model*, i.e. the content value of a variable remains unchanged after a new assignment.

The results indicated that more than half of the students have not comprehended the dynamic meaning of the programming variable. In addition, they held the representation of equality for the assignment statement while facing successive assignments as mathematical relations. These findings are consistent with previous research which indicated that students have difficulties to build adequate representations for the sequence construct of a program and the dynamic change of the values of the variables (Du Boulay, 1986; Rogalski & Vergnaud, 1987; Samurçay, 1989).

The second research question examined if, and to what extent, students' conceptions of programming variable and the assignment statement are mutually related. Our results appeared to confirm the second research question, since the mathematical representation of both constructs is prevalent between students in the sample. We have recorded three different types of faulty mental models of the assignment statement confirming only three of the eleven different mental models reported by Dehnadi & Bornat (2009):

- The *mathematical model* is prevalent among the students; they consider the successive assignments as mathematical equations or calculation formulas rather than as dynamic modifications acting on particular memory locations
- The *accumulator model*, i.e. the right value in the assignment is added to left
- The *equality model*, i.e. an assignment describes the relation between values (left and right side) which are equal.

In addition, the findings are quite similar to the survey of Corney et al. (2011) which reported that one half of university students, in an introductory computing course, could not respond efficiently to a code swapping the values in two variables.

The third research question examined the degree of applicability of SOLO taxonomy as an efficient framework to study and assess students' representations and mental models of the programming variable and the assignment statement. The results of SOLO analysis showed that the majority of the students in the sample tended to manifest prestructural, unistructural and multistructural responses to the research tasks. Despite that studying extended abstract level is a very interesting issue for computer programming educational research, the tasks designed for this particular study did not aim at students' performance in this level. From a research perspective and confirming previous investigations (Lister et al., 2006; 2009; Sheard et al., 2008), the results presented offered significant empirical evidence supporting the assumptions regarding the efficacy of SOLO taxonomy to explain students' mental models in computer programming.

Since this study was based on a convenience sample, generalizations to other educational environments should be done with specific care. However, the research results could be of interest and significance in broader or international educational contexts regarding introductory computer programming.

Implications for practice

The results of this study provided information about students' mental models and representations of the programming variable and the assignment statement. Some of the difficulties reported are intrinsic or inherent to programming but there are also themes for redesigning instruction. The findings could be of value for the design and the successful implementation of teaching interventions aiming at improving students' learning in computer programming.

Traditional teaching approaches appeared to be inefficient to support students' developing of appropriate mental models of the basic programming concepts. Undoubtedly, students as novice programmers require time and they develop their programming knowledge in a slow and progressive way. However, teaching the semantics of the programming language or focusing on the features of a specific language is not adequate to help students develop viable representations of the programming constructs and acquire the necessary programming skills. Teachers should address a pedagogical change in their instruction by emphasizing on *problem-solving* and *algorithm design* patterns rather than on the syntactic details and the features of specific languages. In addition, it is of great importance for instructional practice to systematically engage students into properly designed *code tracing and explaining* activities, in both classroom and laboratory sessions.

Variables, the assignment statement and the sequence structure constitute the basic constructs needed for students to understand the nature of programming. Our findings strongly support the importance of starting with variables in introductory programming to provide students with viable models of the basic programming constructs applicable in problem solving. Only by knowing the basic programming building blocks students will be able to combine programming constructs and integrate code segments into coherent structures and effective algorithms. The approach of *roles of variables* proposed by Sajaniemi & Kuittinen (2005) appears to be a promising framework, easily linked to code and pseudocode programming paradigms.

In conclusion, educators need to place explicit pedagogical emphasis on the design of students' learning activities and the tools used to help students to improve their mental models of programming concepts. The need to develop a coherent instructional framework about the basic programming concepts is still an open research issue. To address this problem we are working on a learning framework integrating inquiry and cognitive conflict along with the simulation of programming concepts and programs (Vrachnos & Jimoyiannis, 2008). The key ideas in this framework are the appropriate design of scaffolding and supporting students' inquiry in order to help them recognise their flawed models and actively transform them into viable representations of the basic programming constructs.

Conclusion

In this paper we have presented an analysis of the representations held by secondary education (K-12) students about the concepts of programming variable and the assignment statement. Students' responses and justifications were mapped using the SOLO taxonomy. The results showed that approximately more than one half of the students in the sample tended to manifest prestructural, unistructural and multistructural representations. In addition, the findings provided evidence that students' programming thinking was determined by three different models about the concept of variable and the assignment statement. The paper ends with two main conclusions: a) SOLO taxonomy constitutes an

efficient schema for research and students' assessment in introductory programming lessons; b) developing a pedagogically coherent instructional framework for introductory programming should start with the concepts of variable and assignment as building blocks.

References

Biggs, J. (1999). *Teaching for quality learning at university: What the student does*. Buckingham: The Society for Research into Higher Education and Open University Press.

Biggs, J. B., & Collis, K. F. (1982). *Evaluating the quality of learning: The SOLO taxonomy (Structure of the Observed Learning Outcome)*. New York: Academic Press.

Bonar, J., & Soloway, E. (1985). Preprogramming knowledge: a major source of misconceptions in novice programmers, *Human-Computer Interaction*, 1, 133-161.

Brooks, R.E. (1990). Categories of programming knowledge and their application. *International Journal of Man-Machine Studies*, 33, 241-246.

CF (1998). *Curriculum Framework*. Athens: Greek Ministry of Education, Pedagogical Institute (in Greek).

Chalk, P., Boyle, T., Pickard P., Bradley C., Jones, R., & Fisher, K. (2003). Improving pass rates in introductory programming. *Proceedings of the 4th Annual LTSN-ICS Conference* (pp. 6-10). NUI Galway.

Chan, C. C., Chui, M. S., & Chan, M. Y. C. (2002). Applying the Structure of the Observed Learning Outcomes (SOLO) taxonomy on student's learning outcomes: an empirical study. *Assessment & Evaluation in Higher Education*, 27(6), 511-527.

Chen, N. S., & Zimitat, C. (2004). Differences in the quality of learning outcomes in a F2F blended versus wholly online course. In R. Atkinson, C. McBeath, D. Jonas-Dwyer & R. Phillips (eds.), *Beyond the comfort zone: Proceedings of the 21st ASCILITE Conference* (pp. 175-179). Perth, Australia. Retrieved 20 May 2010, from http://www.ascilite.org.au/conferences/perth04/procs/chen.html.

Chick, H. L. (1998). Cognition in the formal modes: Research mathematics and the SOLO taxonomy. *Mathematics Education Research Journal*, 10(2), 4-26.

Corney, M., Lister R., & Teague D. (2011). Early relational reasoning and the novice programmer: Swapping as the "Hello World" of relational reasoning. *Proceedings of the 13th Australasian Computer Education Conference (ACE 2011)*, Perth, Australia: Australian Computer Society, Inc.

Corritore, C. & Wiedenbeck, S. (1991). What do novices learn during program comprehension?. *International Journal of Human Computer Interaction*, 3(2), 199-222.

Dehnadi, S., Bornat, R. & Adams R. (2009). Meta-analysis of the effect of consistency on success in early learning of programming. Retrieved 20 May 2010, from http://www.eis.mdx.ac.uk/research/PhDArea/saeed/SD_PPIG_2009.pdf.

de Raadt, M. (2007). A review of Australian investigations into problem solving and the novice programmer. *Computer Science Education*, 17(3), 201-213.

Du Boulay, B. (1986). Some difficulties of learning to program. *Journal of Educational Computing Research*, 2(1), 57-73.

Du Boulay, B., O'Shea, T., & Monk, J. (1989). The black box inside the glass box: presenting computing concepts to novices. In E. Soloway & J.C. Spohrer (eds.), *Studying the novice programmer* (pp. 431–446). Hillsdale, NJ: Lawrence Erlbaum.

Ebrahimi, A. (1994). Novice programmer errors: Language constructs and plan composition. *International Journal of Human-Computer Studies*, 41, 457-480.

Eckerdal, A., McCartney, R., Moström, J. E., Ratcliffe, M., & Zander, C. (2006). Can graduating students design software systems?. *ACM SIGCSE Bulletin*, 38(1), 403-407.

Eckerdal, A., & Thune, M. (2005). Novice Java programmers' conceptions of "object" and "class", and variation theory. *ACM SIGCSE Bulletin*, 37(3), pp. 89-93.

Garner, S., Haden, P., & Robins A. (2005). My program is correct but it doesn't run: a preliminary investigation of novice programmers' problems. *Proceedings of the Australasian Computing Education Conference 2005* (pp. 173-180). Newcastle, Australia: Australian Computer Society, Inc.

Gilmore, D.J. (1990). Expert programming knowledge: A strategic approach. In J.M. Hoc, T.R.G. Green, R. Samurçay & D.J. Gillmore (eds.), *Psychology of Programming* (pp. 223-234). London: Academic Press.

Green, T.R.G. (1990). Programming languages as information structures. In J.M. Hoc, T.R.G. Green, R. Samurçay & D.J. Gillmore (eds.), *Psychology of Programming* (pp. 117-137). London: Academic Press.

Hazel, E., Prosser, M., & Trigwell, K. (2002). Variation in learning orchestration in university biology courses. *International Journal of Science Education*, 24(7), 737-751.

Hoc, J.-M., Green, T. R. G., Samurçay, R., & Gilmore D. J. (1990). *Psychology of Programming*. London: Academic Press.

Holland, S., Griffiths, R., & Woodman, M. (1997). Avoiding object misconceptions. *ACM SIGCSE Bulletin,* 29(1), 131-134.

Hundhausen, C., & 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 and Computing*, 18(1), 22-47.

Jimoyiannis, A. (2000). Teaching computer programming in secondary education. Students' difficulties and perceptions of the concept of programming variable. *The Base* (vol. 2, pp. 35-42). Ioannina (in Greek).

Jimoyiannis, A., & Komis, V. (2000). The concept of variable in programming: Lyceum students' difficulties and misconceptions. In V. Komis (ed.), *Proceedings of the 2nd Pan-Hellenic Conference "Information and Communications Technologies in Education"* (pp. 103-114). Patra, Greece (in Greek).

Jimoyiannis, A., & Komis, V. (2004). The study of Lyceum students' representations of the computer data flow and the role of the basic computer units. In P. Politis (ed.), *Proceedings of the 2nd Pan-Hellenic Conference "Didactics of Informatics"* (pp. 73-85). Volos, Greece (in Greek).

Lane, H. C. &, VanLehn, K. (2005).Teaching the tacit knowledge of programming to novices with natural language tutoring. *Computer Science Education*, 15(3), 183-201.

Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, E., Sanders, K., Seppälä, O., Simon, B., Thomas, L., (2004). A multi-national study of reading and tracing skills in novice programmers. *ACM SIGCSE Bulletin*, 36(4), 119-150.

Lister, R., Simon, B., Thompson, E., Whalley, J. L., & Prasad, C. (2006). Not seeing the forest for the trees: novice programmers and the SOLO taxonomy. *ACM SIGCSE Bulletin*, 41(3), 118-122.

Lister, R., Fidge, C., & Teague, D. (2009). Further evidence of a relationship between explaining, tracing and writing skills in introductory programming. *ACM SIGCSE Bulletin*, 41(3), 161-165.

Lister, R., Clear, T., Simon, Bouvier, D. J., Carter, P., Eckerdal, A., Jacková, J., Lopez, M., McCartney, R., Robbins, P., Seppälä, O., & Thompson, E. (2009). Naturally occurring data as research instrument: Analyzing examination responses to study the novice programmer. *ACM SIGCSE Bulletin*, 41(4), 156-173.

Ma, L., Ferguson, J., Roper, M., Ross, I., & Wood M. (2009). Improving the mental models held by novice programmers using cognitive conflict and Jeliot visualisations. *ACM SIGCSE Bulletin*, 41(3), 166-170.

Ma, L., Ferguson, J., Roper, M., & Wood, M. (2011). Investigating and improving the models of programming concepts held by novice programmers. *Computer Science Education*, 21(1), 57-80.

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B., et al. (2001). A multinational, multi-institutional study of assessment of programming skills of first-year CS students. *ACM SIGCSE Bulletin*, 33(4), 125-180.

McGettrick, A, Boyle, R, Ibbett, R, Lloyd, J, Lovegrove, L, and Mander, K. (2005). Grand challenges in computing education – A summary. *The Computer Journal*, 48(1), 42-48.

Moreno, A., Myller, N., Sutinen, E., & Ben-Ari, M. (2004). Visualizing program with Jeliot3. *International Working Conference on Advanced Visual Interfaces* (pp. 373-380).

Naps, T., Eagan, J.R., & Norton, L.L. (2003). JHAVE – An Environment to Actively Engage Students in Web-based Algorithm Visualizations. *31st SIGCSE Technical Symposium on Computer Science Education* (pp. 109-113).

Palumbo, D. B., & Reed, W. M. (1991). The effect of BASIC programming language instruction on high school students' problem-solving ability and computer anxiety. *Journal of Research on Computing in Education*, 3, 343-372.

Padiotis, I., & Mikropoulos, T. A. (2010). Using SOLO to evaluate an educational virtual environment in a technology education setting. *Educational Technology & Society*, 13 (3), 233–245.

Perkins, D. N., Hancock, C., Hobbs, R., Martin, F., & Simmons, R. (1989). Conditions of learning in novice programmers. In E. Soloway & J. C. Spohrer (eds.), *Studying the Novice Programmer* (pp. 261-279). Hillsdale, NJ: Lawrence Erlbaum.

Postner, L., & Stevens, R. (2005). What resources do CS1 students use and how do they use them?. *Computer Science Education*, 15(3), 165-182.

Putman, R., Sleeman, D., Baxter, J. & Kuspa, L. (1989). A summary of misconceptions of high-school BASIC programmers. In E. Soloway & J. C. Sprohrer (eds.), *Studying the Novice Programmer* (pp. 301-314). Hillsdale, NJ: Lawrence Erlbaum.

Rist, R. S. (1989). Schema creation in programming. *Cognitive Science*, 13, 389-414.

Rist, R. S. (1991). Knowledge creation and retrieval in program design: A comparison of novice and intermediate student programmers. *Human-Computer Interaction*, 6, 1-46.

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

Rogalski, J., & Samurçay, R. (1990). Acquisition of programming knowledge and skills. In J.M. Hoc, T.R.G. Green, R. Samurçay, & D.J. Gillmore (eds.), *Psychology of Programming* (pp. 157-174). London: Academic Press.

Rogalski, J., & Vergnaud, G. (1987). Didactique de l'informatique et acquisitions cognitives en programmation. *Psychologie Française*, 32(2), 267-273.

Sajaniemi, J. (2002). An empirical analysis of roles of variables in novice-level procedural programs. *Proceedings of IEEE 2002 Symposia on Human Centric Computing Languages and Environments* (pp. 37-39).

Sajaniemi, J., & Kuittinen, M. (2005). An experiment on using roles of variables in teaching introductory programming. *Computer Science Education*, 15(1), 59-82.

Samurçay, R. (1989). The concept of variable in programming: Its meaning and use in problem solving by novice programmers. In E. Soloway & J.C. Spohrer (eds.), *Studying the novice programmer* (pp. 161–178). Hillsdale, NJ: Lawrence Erlbaum.

Sleeman, D., Putnam, R., Baxter, J., & Kuspa, L. (1986). Pascal and high school students: A study of errors. *Journal of Educational Computing Research*, 2(1), 5-23.

Sheard J., Carbone A., Lister R., Simon B., Thompson E., & Whalley J. L. (2008). Going SOLO to assess novice programmers. *ACM SIGCSE Bulletin - ITiCSE '08*, 40(3), 209-213.

Simon, Fincher, S., Robins, A., Baker, B., Box, I., Cutts, Q., de Raadt, M., Haden, P., Hamer, J., Hamilton, M., Lister, R., Petre, M., Sutton, K., Tolhurst, D., & Tutty, J. (2006). Predictors of success in a first programming course. *Eighth Australasian Computing Education Conference*, *ACE2006* (pp. 189-196). Hobart, Tasmania: Australian Computer Society.

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

Spohrer, J.C., & Soloway, E. (1989). Novice mistakes: Are the folk wisdoms correct? In E. Soloway & J.C. Spohrer (eds.), *Studying the novice programmer* (pp. 401–416). Hillsdale, NJ: Lawrence Erlbaum.

Soloway, E. (1986). Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM*, 29(9), 850-858.

Soloway, E., & Spohrer, J.C. (1989). *Studying the novice programmer*. Hillsdale, NJ: Lawrence Erlbaum.

Thomson, E. (2007). Holistic assessment criteria: applying SOLO to programming projects. *Proceedings of the 9th Australian Computer Society* (pp. 155-162). Ballarat, Victoria: Australian Computer Society.

Vrachnos, E. & Jimoyiannis, A. (2008). DAVE: A Dynamic Algorithm Visualization Environment for novice learners. *Proceedings of the 8th IEEE International Conference on Advanced Learning Technologies* (pp. 319-323), Santander, Spain: IEEE.

Whalley, J., Lister, R., Thompson, E., Clear, T., Robbins, P., & Prasad, C. (2006). An Australasian study of reading and comprehension skills in novice programmers, using the Bloom and SOLO Taxonomies. *Proceedings of the Eighth Australasian Computing Education Conference (ACE2006)* (pp. 243-252). Retrieved 20 May 2010, from http://crpit.com/confpapers/CRPITV52Whalley.pdf.

Wiedenbeck, S., & Ramalingam, V. (1999). Novice comprehension of small programs written in the procedural and object-oriented styles. *International Journal of Human-Computer Studies*, 51, 71-87.

Wiedenbeck, S., Fix, V., & Scholtz, J. (1993). Characteristics of the mental representations of novice and expert programmers: An empirical study. *International Journal of Man-Machine Studies*, 39, 793-812.

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

To cite this article: Jimoyiannis, A. (2011). Using SOLO taxonomy to explore students' mental models of the programming variable and the assignment statement. *Themes in Science and Technology Education*, 4(2), 53-74.

URL: http://earthlab.uoi.gr/theste