Introducing Secondary Education Students to Algorithms and Programming

VASSILIOS DAGDILELIS
Department of Educational and Social Policy, University of Macedonia, 156, Egnatia str., P.O. Box 1591, 54006, Thessaloniki, Greece
E-mail: dagdil@uom.gr

MAYA SATRATZEMI and GEORGIOS EVANGELIDIS
Department of Applied Informatics, University of Macedonia, 156, Egnatia str., P.O. Box 1591, 54006, Thessaloniki, Greece
E-mails: {maya,gevan}@uom.gr

Abstract

In Greece, the development of the teaching of Information Technology (IT) in schools has been greatly influenced by the rapid development of technology, making *IT literacy* a priority for all individuals. Consequently, the teaching of algorithms and programming, with the ulterior motive of teaching modeling as well as problem solving, has been greatly limited in Secondary Education. However, we strongly believe that algorithms and programming constitute an important intellectual tool and should be included in basic education. In any case, both the research literature findings, as well as, our own experience confirm the fact that novice programmers come up against many *mental obstacles* in their attempts to understand the functioning of programs or the construction of algorithms. In order to deal with these difficulties and successfully teach the elementary concepts of algorithms and programming, we have developed *didactic scenarios,* which are based on specially designed educational software. In conjunction with this, we are attempting to develop a program for the systematic training of those students who will become IT teachers in Secondary or Primary Education. The most significant findings of our research are summarized as follows: (a) The development of educational software and its experimental use in the teaching process allow us to formulate several general rules related to the specific didactic characteristics, which these environments should include. (b) Educational software is effective only when it is incorporated within the framework of the didactic scenarios that the teacher organizes and which are supported by the software.
(c) Teachers do not spontaneously use educational software in the context of this rationale. Therefore, specific training is required so that they adopt and use such software in didactic scenarios.

Keywords: algorithms, educational programming environments, didactic scenarios, secondary education, computer science education research

1. Introduction

Information Technology (IT) is an undisputed reality in education: it has become widely accepted in schools but also it appears that there exists a demand for *more IT* both from the state (educational system) and the citizens (parents). In Greece, the use of term IT in the above sentence does not usually imply an introduction to the science of information technology, but it rather stands for *information technology literacy*, or in other words, the

set of skills and knowledge that nowadays comprise a standard vocational prerequisite and a requirement for a citizen to be considered an active member of the society.

IT in education or *educational IT* appears in many facets:

(i) it can be used as a *teaching subject* (i.e., computer science education),
(ii) it can comprise a *teaching tool* (i.e., using IT to teach various subjects), or
(iii) it can indirectly but significantly *influence the teaching process* (i.e., when computer networks are used to help students in remote settings collaborate and interact with each other – Computer Mediated Communication).

Usually, each one of the above facets can have many variants. For example, IT as a teaching subject can imply either plain IT literacy or the teaching of programming in the framework of vocational education or even anything in between.

In this paper we refer to an important facet of educational IT: the teaching of algorithms and programming to novices. This facet of educational IT exists in all educational levels, but at least in the Greek educational system, it is undervalued. Despite that, we strongly believe that it comprises an invaluable skill for students, even when they do not plan to become IT professionals or scientists.

In the following sections we will attempt to analyze the framework of our research in the teaching of algorithms and programming in secondary education and briefly report on some results we have obtained. The theoretical framework of our research lies in the area of the Didactics of Informatics, in a close analogy to the corresponding theoretical framework of the Didactics of Mathematics (Sections 2 and 3). Our research findings confirm the fact that the teaching of algorithms and programming poses significant mental problems to many novice student programmers. We demonstrate some examples of these problems that we believe can be at least partially overcome by using specially designed and implemented *didactic scenarios* (Section 4). To realize these scenarios we often use educational software some of which we have developed. The design and the pilot or systematic use of such software allows us to define some specific characteristics that it should possess in order to effectively support the teaching process (Section 5). We formulate the hypothesis that teachers do not feel comfortable with or accept this raison d'être – they believe that mere knowledge of the subject of algorithms and programming is sufficient for its effective teaching. This situation appears to be unproductive (since students continue to face the same problems) and calls for the design and implementation of a special training program (Section 6). We finally present our conclusions (Section 7).

2. Educational IT and Its Evolution

In most European school systems, IT was introduced about 20 years ago: it was in the early 80s that computers entered the classroom, mainly in secondary education (see for example (Baron, 1989)). The early years of educational IT are characterized by a dedication to the teaching of the way computers operate and of the elements of programming. But, in the

last decade, one can observe a shift in the teaching subject of educational IT from the acquisition of knowledge on how a computing system works and how it can be programmed to the knowledge on how a computing system can be used and be embodied in everyday life. The notion of *user* becomes fundamental in this shift.

The invasion of IT in the world of economy has gradually created the need for the acquisition of elementary computing skills by each and every one individual. This is the driving force behind another shift that can also be observed: the point of reference for the teaching of IT is not the *science of IT per se* but the *social practices with respect to IT*, that is, the ways in which IT is being embodied in society.

Some of the uses of computers are equally important for the individual and the society. The basic skills of managing and exploiting digital information are integral elements of contemporary *IT literacy*, which is now considered not only an essential but also a necessary skill for all individuals. IT literacy is a new entity in the educational systems; it is almost autonomous but interdisciplinary, and evolves with a fast pace. Some recent examples of its evolution are the notions of *effective searching for information (or "googling")*, *e-democracy*, and *secure web browsing*, which now constitute knowledge, skills and meta- skills that must be part of the compulsory education for all individuals.

A relative downgrading of the activities related to algorithms and programming accompanied this shift in IT literacy. This phenomenon was particularly evident in Greece, be- cause in addition to the international trends, the Greek educational system had certain peculiarities that intensified it. These peculiarities have also influenced the evolution of educational IT: for example, even today the upper levels of secondary education still fol- low the classic structured programming paradigm (Pascal) and have not adopted the OOP paradigm. This is a fact that also limits and directs our research efforts in secondary IT education; our research concerning OOP is mainly oriented towards higher education students. Despite its conservative nature, the idiosyncratic status of educational IT in Greece, does favor innovative and experimental teaching – of course in the context of a predetermined framework.

Our research is basically directed towards the teaching of algorithms and programming. Despite the relative downgrading of these two subjects of educational IT that we mentioned above, we consider them of utmost importance in IT literacy, as a scientific or professional choice, and, most importantly, as a mental tool for attacking and solving various problems. We believe that algorithms and programming are *invaluable general-purpose tools in the problem solving process* and this is because, as C. Hoyles mentions, "if you wish to learn something well teach it to others, and if you want to learn it perfectly teach it to a computer". This is because in a computing environment it is absolutely necessary to formally and completely define the entities and relationships of a problem, and this is because *this is how the machine works* and not because there exists a certain *didactical contract* (Brousseau, 2000) or because this is the will of the instructor (Snyder, 1999).

Viewing programming as such a special mental tool, also justifies our theoretical frame- work that we illustrate in the following section.

3. Theoretical Framework of Our Research and Student Conceptions

The research we conduct regarding the teaching of algorithms and programming in secondary (and higher) education is associated with a certain underlying principle or *rationale*, that is, a theoretical reference framework, which evolves in parallel with the various empirical research results we obtain. Thus, our research hypotheses and the way we interpret our findings are associated with a rationale that is also being revised depending on the results obtained each time. This theoretical framework also guides the design, implementation and use of the educational software we develop. Briefly, we should describe our theoretical framework as some kind of *constructivism* – a framework recently accepted even for the teaching of Informatics (Mordechai, 2001; Greening, 1999): we believe that students construct their knowledge by interacting with their environment in the context of didactic scenarios. The collaboration and interaction with the other participants (teacher, class fellows) can play a critical role in their improvement. So, we do not consider constructivism only as a general pedagogical theory, but, instead, we use it as a framework for the construction of didactic scenarios, following the paradigm of Didactics of Mathematics (Brousseau, 2000).

As a starting point of our research, we consider the errors students make and the difficulties they face during problem solving. Of course, a large percentage of students make random errors; they do not have difficulties in understanding an algorithm and solving a problem. We refer to an equally important, if not larger, percentage of students that make systematic errors and have difficulties in coping with algorithms and programming.

Algorithms and programming can be taught in many different ways and levels of detail. However, regardless the teaching methodology, a significant percentage of novice student programmers have major difficulties in understanding basic algorithms and solving programming problems – even the simplest ones (Spohrer *et al.*, 1985; Spohrer and Soloway, 1986). This observation appears to have some invariant elements and one can formulate a strong hypothesis that these difficulties are in reality the result of the *conceptions* students have, i.e., in our case the possibly erroneous, incomplete or even inconsistent knowledge they possess and use during problem solving.

If the hypothesis that students have certain conceptions is true, then these conceptions should be expressed in various ways, that is, in more than one different context. Our observations show that in many cases student actions have a common origin – this is what we call *student conceptions*.

A classic example of this kind is the problem of computing the area of a rectangle. Many novice student programmers believe that the following code computes the area of a rectangle:

```
area := base*height;
read(base);
read(height);
write(area);
```

They are really surprised to realize that the area is not computed correctly (Dagdilelis *et al.*, 2003). Our findings confirm older results of (Pea, 1986) where this example appeared for the first time. This kind of *parallelism* (term introduced by R. Pea) is quite common.

The error comes form the fact that students do not take into account the order in which the statements of the program are executed – they just establish that all the required data is present (lengths and the relevant formula).

Another category of errors appears to have a common origin with the one mentioned above, that is, the fact that novice student programmers do not interpret programs in the formal way (according to the meaning of each statement) but in a broader way. Sleeman *et al.* (1987) discovered that students make errors like the following:

```
if <a> then <b>; <c>;
```

is interpreted as

```
if <a> then <b> else <c>;
```

This kind of error can be observed in cases like the following:

```
if age > 35 print "you are too old";
print "you are still young";
```

This systematic error can be attributed to the fact that students possibly interpret this program as *a communication among humans*; thus, it is not necessary to explicitly describe all possible cases.

Our findings show that many students have difficulty with the use of input statements, and more specifically the statement *read* or the equivalent ones in the various programming languages. Therefore, young students (age of 15) make errors like the following. When they have to compute a number, assign it to a variable, say A, and then print the value of A, they use an extra statement of the type read(A):

```
read(x);
read(z);
A := x+z;
read(A);
write(A);
```

When asked why they use this statement, a common answer is that "(the computer) has to read the value from the memory" before printing it. In this situation, the read statement is not interpreted in the formal way, but in a broader way, as if the computer has to see the value of the variable before printing it, like a human would do. Many novice programmers – even freshmen university students – make similar errors, and this is an indication that the origin of these errors has nothing to do with the level of maturity of the students.

In the above examples, students conceive their communication with the computer as if it were a communication with some other human being, or they do not take into account the order statements should be executed. In all cases, the common observation is that students instead of following the formal rules that apply to the programming environment they use, they follow rules that govern social relationships or rules that apply to other contexts. In this case, teaching should focus on the refutation of this conception and its replacement by some other more functional one.

The relevant research literature of the past 20 years reports many such systematic errors made by novice student programmers (the *Workshops of Empirical Studies of Programmers* and the *Psychology of Programming Interest Group* constitute valuable resources of this

kind of findings); these are the errors we relate to our hypothesis regarding the conceptions of students.

Here we should pinpoint the fact that despite the extensive research, the obtained results are not integrated in the context of a unified and commonly accepted theoretical framework or rationale. Consequently, many reported results are anecdotal and are of little value to a new teacher (Dagdilelis *et al.*, 1999).

The comprehension difficulties students have in algorithms and programming are not attributed only to the misinterpretation of the semantics of the various program statements. Novice student programmers also have difficulties in devising new algorithms, choosing an appropriate loop statement, managing variables, and in many more areas.

Students have difficulties when trying to address certain categories of problems. One such category is those problems where the "coded" solution greatly differs from the "mental" solution and the latter one cannot be turned into a "coded" solution. A typical example of this category of problems is the computation of the sum of natural numbers from 1 through N, that is, $1 + 2 + 3 + \cdots + N$. Although it appears to be a simple problem, a large percentage of novice programmers cannot solve it. The reason they fail becomes evident when one views the code fragments they produce:

```
x := 1;
x := x+x+1;
x := x+x+1;
```

The same group of students when working in pairs describe their algorithm as following: "you get the first number and then you add it to its successor and you go on in a similar manner .. .". Of course, they soon realize that this strategy leads nowhere since what they get is the series of numbers 1, 3, 7, 15. It appears that their conceptions cannot make them realize the necessity of using a counter and an adder, and they just try to devise solutions similar to the "mental" one. In order to overcome the difficulties they meet, students usually reach a dead end trying to rephrase their "mental" solution into a "more correct" programming solution, as shown below:

```
x := x+x+1;
...
x := x;
x := x+1;
...
x := 2*x+1;
...
```

It appears that students need a few months of practicing before they become capable of solving problems of this kind.

Another category of problems arises when a variable that can be used in more than one situation. For example, some students try to solve the previous problem by using an additional counter:

```
for i := 1 to N do
begin
  counter := counter+1;
```

```
  sum := sum+counter;
end
```

Here is another example: one can choose a number among the first N natural numbers, i.e., 1, 2, 3, … , N , and we wish to count how many times a number has been chosen. Usually, students prefer solutions that do not exploit all the provided information, as demonstrated below,

```
read(preference);
if preference = 1 then counter[1] := counter[1]+1 else ...
```

instead of the more elegant and cheap one:

```
read(preference);
counter[preference] := counter[preference]+1;
```

In these cases, novice programmers devise correct algorithms, but their solutions reveal that they have not mastered the role and the potential usage of the various program variables. It may be the case that exploitation of all the available information requires some kind of intellectual capability that is acquired with experience. Nevertheless, many advanced level programmers in higher education cannot effectively use data structures in their programs.

4. Didactic Scenarios

The results we mentioned in the previous section guide us in the development of specific didactic scenarios and educational software for the confrontation of the difficulties students meet. Specifically, in order to introduce students to the fundamental principles of programming and to help them revise their misconceptions, we make systematic use of didactic scenarios, that is, educational material organized in such a way so that we achieve effective teaching. The basic elements of these didactic scenarios are the tool usefulness and the richness of interactivity the students enjoy.

The first element implies that every newly introduced concept or relationship should comprise a kind of problem solving tool. For example, in order to teach loop statements, the teacher should suggest problems that require the use of such statements, such as the computation of the average GPA of (an unknown number of) students. Appropriate problems should also be suggested in order to clarify the criteria that favor the use of one loop statement over the other (e.g., `for` versus `while` versus `repeat`).

In a similar manner, a teacher should choose problems that show off the usefulness of subroutines. For example, the basic routine for computing the minimal element in a collection can be systematically reused for solving many similar problems.

The choice of the appropriate problem is an essential consideration for successfully introducing new concepts and methods to students. Of equal, if not grater, importance is the creation of educational environments that favor a high degree of interactivity among students. There should be interactivity not only in the educational material (e.g., educational software) but in the rest participants in a didactic scenario, as well. So, students should not only use supporting educational software (we refer to that in the next section) but they

should also work in groups, with each group producing a single solution, possibly acceptable by all its members. The teacher should intervene only in the case a group cannot reach an agreement. Also, in some cases, groups or individuals should present their solution in class and defend its correctness or efficiency.

It is true that organizing a course in such a manner is a hard and, often, a time consuming thing to do. But, the data we have collected so far (Dagdilelis *et al.*, 2002) indicate that it helps students get involved in fruitful interactions that greatly enhance their ability to master new concepts and methods.

5. Educational Environments

Novice student programmer misconceptions have some common characteristics:

- They come into view in a systematic way (i.e., independently of the teaching method).
- They are very hard to change.
- They comprise knowledge that is valid in some fields but not in programming.

We view these (mis-)conceptions as cognitive obstacles. In such situations the use of special educational software can be very helpful. These kinds of software provide real time information about the system (in our case the programming environment) and can address the major demand of novice student programmers, i.e., exactly how a computer executes a program and what the various statements/commands mean.

During the past years, a great deal of educational software has been developed for the teaching of algorithms and programming or more general subjects, such as numbering systems, logic circuits, computer networks, etc. Most of the systems that teach programming offer simplified programming environments and visualization capabilities for the intermediate phases of the execution of a program. We should emphasize the fact that the basic ingredient in a successful teaching experience is not educational software or any other kind of supporting educational material per se, but the way this material is being utilized. Even the most advanced and fancy educational software cannot guarantee a successful teaching experience if it is not incorporated in an appropriate teaching framework. Thus, educational software has a didactic value whose potential has to be unveiled by the teachers that use it.

As we mentioned above, educational software for novice programmers offer simplified programming environments. For example, they incorporate a reduced statement set and use simplified versions of I/O statements (or sometimes no I/O statements). DELYS (Dagdilelis *et al.*, 2003), an educational software environment for teaching computing science to secondary education students, as well as, WIPE (Efopoulos *et al.*, 2003), the web version of DELYS, have students program in a subset of Pascal where variables need not be declared and data input and output is automatically handled by the system.

In order to design these environments we followed certain basic principles that appear to be important.

The first one has to do with the capability students must have to express the solution they devise without having the system interfere by correcting possible errors. Thus, students discover their errors when they reach a dead end situation, a fact that reveals that

there exists a cognitive obstacle between their conceptions and reality. So, for example, in the various environments that we have created in DELYS and deal with the translation between the various numbering systems, students are allowed to make errors or try erroneous algorithms but they eventually reach a dead end situation and are forced to revise their method.

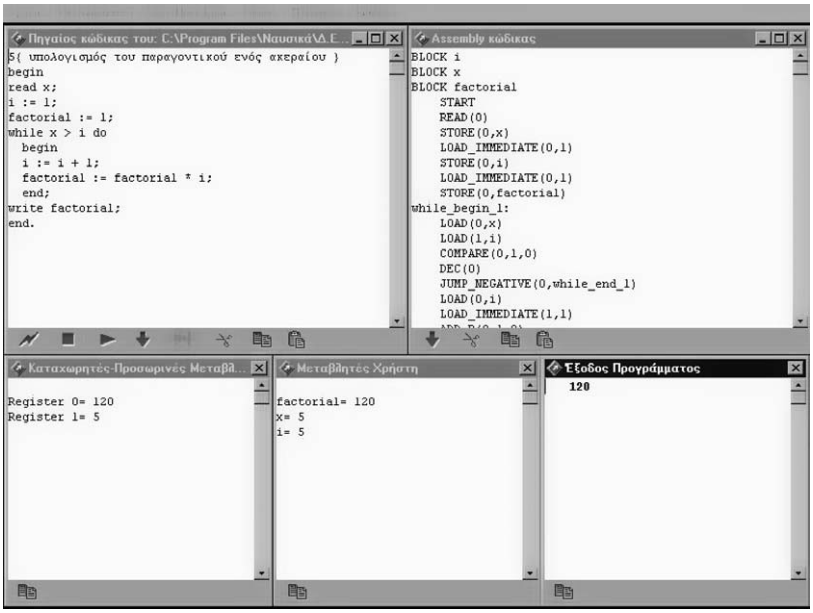
The second principle that we followed has to do with the provision of information about the system (educational environment) to the user (student) during its operation. The understanding of the operation of a complicated mechanism, like program execution, can be facilitated when there is a provision to observe the details of the state of the system. This feature can be implemented in various levels of detail and in different representation ways that can operate in parallel or independently. Both in DELYS and WIPE that were specifically created for secondary education students, as well as in other similar software that we have up till now used with higher education students – AnimPascal (Satratzemi et al., 2001), AnimGraph and Didagraph (Dagdilelis *et al.*, 1998; Dagdilelis and Satratzemi, 1998), AnimGraph (Satratzemi, 2000), ObjectKarel (Xynogalos and Satratzemi, 2002), Post's machine (Dagdilelis *et al.*, 2001) – execution of a program can be observed at the source code level (by having the currently executed command/statement highlighted), and, depending on the software, one can observe the state of assembly code, the intermediate values of variables, the contents of machine registers, and, of course, the program output (see Figure 1).

Users cannot only observe the execution of the program, but in some occasions they can also intervene (for example, in DELYS, they have the ability to modify the automatically produced assembly code that corresponds to the original source code).

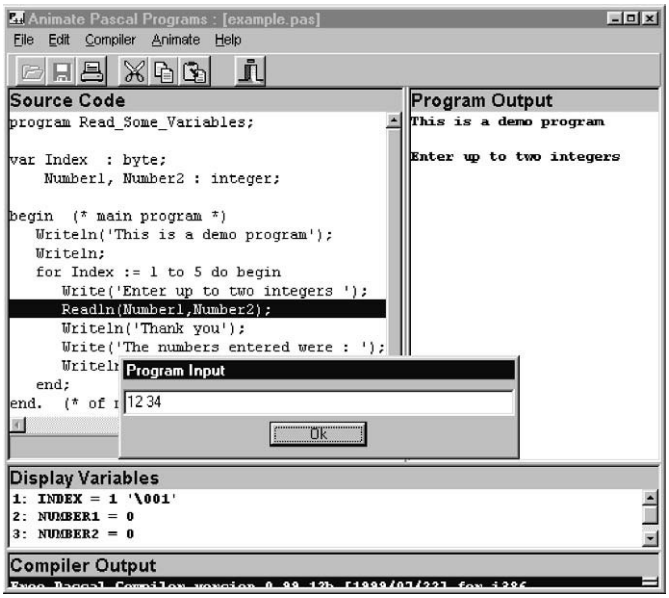
Such features provide great didactic aids to the teacher who can demo, explain, reproduce certain programming situations, and create various educational activities for the students that explore the elementary programming concepts through the use of these environments. A number of independent qualitative studies (Ziveldis and Kappas, 2003) confirm our findings that our educational software helps students understand program execution by computers and stop considering it a black box process.

Another technique for implementing such educational environments that simplifies the presentation and reduces the complexity of the concept to be taught is the use of analogies and metaphors. We refer to the creation of virtual environments with familiar everyday life objects where students can be initiated to various Computing Science concepts.

Such an example is the Virtual Scale (*http://macedonia.uom.gr/~delys/SoftSample.htm*), a pair of scales, where students can place "decimal" weights on one scale and "binary" ones on the other (see Figure 2). The Virtual Scale allows teachers to easily explain the difference between a number (a weight) and its representation (a collection of weights), a concept that students usually have a hard time to understand since they equate numbers to their decimal representation. The Virtual Scale can also help teachers deal with issues such as the properties of the numbering systems and the role of symbols and representations. For example, students that claim that they have understood the way the system works, temporarily leave the classroom and when they return they are presented with various messages. Message "317" implies that weights corresponding to 317 grams have been positioned on the decimal part of the scale, and message "101101" implies that the binary



(a)



(b)

Figure 1. Visualizing execution in (a) DELYS and (b) AnimPascal.

Figure 2. The virtual scale.

weights 32, 8, 4, 1 that correspond to 45 grams have been positioned on the binary part of the scale. Ambiguous messages, like "101", are excellent grounds for a discussion on the symbols used to represent numbers (Dagdilelis *et al.*, 2003).

We are currently designing a similar system for smoothly introducing students to programming through an environment that manages (directly and via programming) the matrix board of a stadium.

A very important element in programming environments is the quality of *the interaction between the student and the computer*. It is very important that the messages generated by the system are comprehensible and accurately pinpoint the errors students make (see Figure 3). It has been found that novice programmers usually have difficulties with the syntax of their programs (Dagdilelis *et al.*, 2003).

Various syntax editors have been proposed. Post's Machine and ObjectKarel use menu lists that help students choose the desired statement (see Figure 4). Thus, they can produce code that is always syntactically correct. We have obtained results from pilot uses of ObjectKarel that show that students favorably receive such features (Xynogalos and Satratzemi, 2001).

6. Teacher Training

The teaching of programming requires that teachers have a broad knowledge on programming issues (Gal-Ezer and Harel, 1998). It is important that teachers are trained not only on the subject of programming, but also on the difficulties related to its teaching.

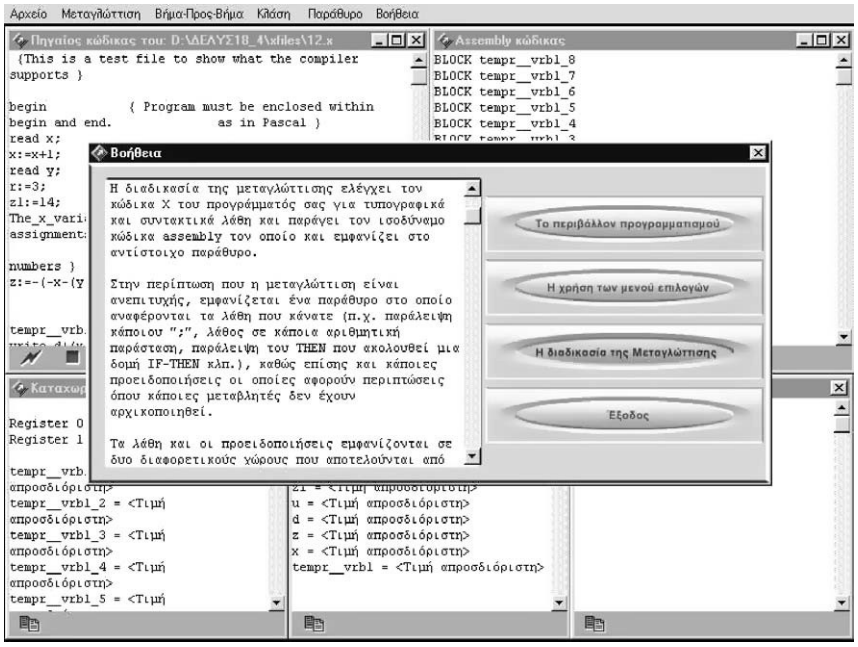
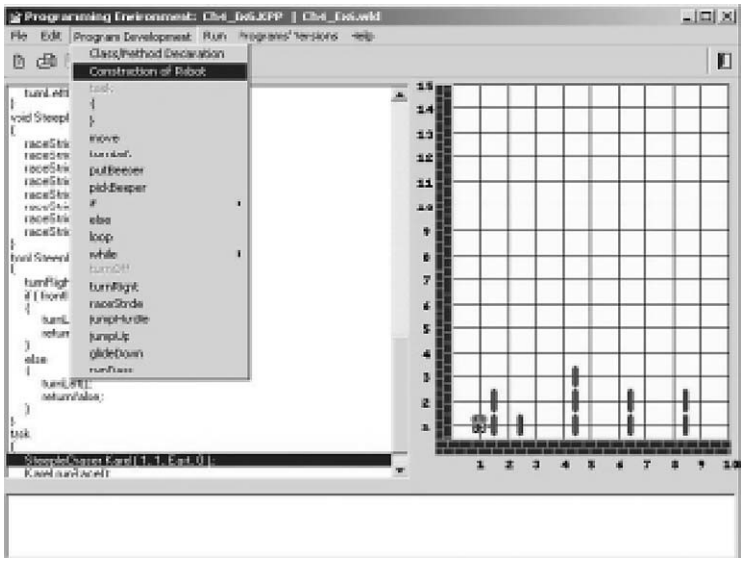


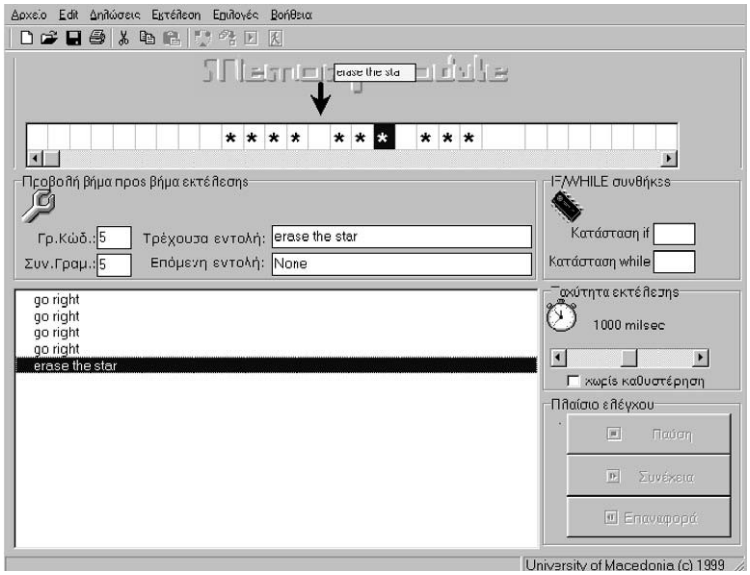
Figure 3. Messages to the user in DELYS.

As far as the Greek educational system is concerned, in addition to the graduates of Computer Science and Information Technology departments that may choose to teach programming, there are additional categories of teachers, e.g., primary school teachers, which may have to teach IT literacy or more theoretical IT courses (that include the teaching of elementary programming concepts).

In the courses we teach in our departments, we teach programming to our students and potential teachers by incorporating most of the techniques we described in the previous section for the secondary education students, i.e., the choice of appropriate problems with didactic value and the use of programming environments that favor the collaboration and interaction among students. Regarding the pedagogic and didactical preparation of teachers, it appears that they tend to reproduce the didactic model they have been trained with. It is the determinant for their pedagogical choices when designing and implementing their courses. As an example, teachers tend not to use the various educational software they are provided with in the way they are supposed and designed to be used; they just incorporate them in the typical context of the course they teach. We believe that teachers must receive special training so that they adopt alternative pedagogical practices. Future teachers are often convinced that basic knowledge of the subject to be taught is sufficient for an effective teaching of the subject in question. This is attributed to their limited teaching experience and constitutes a hard to address didactic problem.



(a)



(b)

Figure 4. Menus for insertion of instructions in (a) Post's Machine and (b) ObjectKarel.

7. Conclusion

The dramatic advancement of IT during the recent years (emergence of new technologies, IT literacy becoming a requirement for all work-force) has caused a minor retreat in the teaching of algorithm and programming in secondary education where they traditionally had a strong presence.

The past twenty years there have been conducted many research efforts related to the teaching of the concepts of programming. Despite that, neither a unifying theoretical framework similar to the ones found in other scientific knowledge areas, nor a research paradigm to guide the research, pose the appropriate questions to be asked, interpret the findings and guide the teaching and the design of educational software environments have been established. Thankfully, concepts and methods from scientifically related knowledge areas offer a framework that can be adopted and adapted.

We have identified many of the difficulties that arise during the teaching of programming. Novice student programmers don't use the various commands/statements of the programming languages in accordance to their definition and find it hard to comprehend some of the important programming concepts. We proposed some unifying concepts, such as the student conceptions and misconceptions, that cover a wide range of the systematic errors made by novice student programmers.

To overcome and refute these conceptions and to introduce new programming concepts to students, we propose the use of didactic scenarios, that is, educational material organized and based on concepts like the use of supplementary educational software and the strong collaboration and interaction among students. Our findings demonstrate that such courses offer students a quite fruitful learning experience.

Educational software specifically designed for introducing novices to programming is also quite helpful in teaching programming concepts to students. Such software is characterized by graphical user interface simplicity, user friendliness in the interaction with the students and the advanced visualization capabilities.

Other important characteristics are the provision of information and knowledge in multiple representations and the ability of the users to express their conceptions even when they are erroneous. The educational environments are often implementing virtual environments – with objects students are familiar with and can directly manipulate – and support a simplified management of the various programming and other IT concepts.

As far as the education and training of potential teachers of IT in secondary education is concerned, we believe that their didactical and pedagogical training is a must, since they usually adopt a conservative attitude and tend to have a teaching philosophy that reproduces their own experiences.

References

Baron, G. L. (1989) L'informatique discipline scolaire? *Le Cas des Lycees*. PUF Editions, Paris, France.

Brousseau, G. (2000) *Theory of Didactical Situations in Mathematics: Didactique des Mathématiques, 1970–1990*. Kluwer Academic, Dordrecht.

Dagdilelis, V. and Satratzemi, M. (1998) DIDAGRAPH: Software for teaching graph theory algorithms. In *Proceedings of the 3rd Annual Conference on Innovation and Technology in Computer Science Education, ACM SIGCSE Bulletin*, **30**(3).

Dagdilelis, V. and Satratzemi, M. (2001) Post's Machine: A didactic microworld as an introduction to formal programming. *Education and Information Technologies*, **6**(2), 123–141.

Dagdilelis, V., Satratzemi, M., and Evangelidis, G. (1999) Didactics too, not only technology. In *Proceedings of the 4th Annual Conference on Innovation and Technology in Computer Science Education, ACM SIGCSE Bulletin*, **31**(3).

Dagdilelis, V., Satratzemi, M., and Evangelidis, G. (2002) What they really do? Attempting (once again) to model novice programmers' behavior. In *Proceedings of the 7th Annual Conference on Innovation and Technology in Computer Science Education, ACM SIGCSE Bulletin*, **34**(3).

Dagdilelis, V., Evangelidis, G., Satratzemi, M., Efopoulos, V., and Zagouras, C. (2003) DELYS: A novel microworld-based educational software for teaching Computer Science subjects. *Computers & Education*, **40**(4), 307–325.

Efopoulos, V., Evangelidis, G., and Dagdilelis, V. (2003) WIPE (Web Integrated Programming Environment) – design principles and architecture. In *Proceedings of the 9th Panhellenic Conference on Informatics* (to appear), Thessaloniki, Greece, November.

Gal-Ezer, J. and Harel, D. (1998) What (else) should CS teachers know? *Communications of ACM*, **41**(9).

Greening, T. (1999) Emerging constructivist forces in computer science education: Shaping a new future? In *Computer Science Education in the 21st Century*, T. Greening (ed.). Springer, Berlin.

Mordechai, Ben-Ari (2001) Constructivism in computer science education. *Journal of Computers in Mathematics & Science Teaching*, **20**(1).

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

Satratzemi, M. (2000) AnimGraph: A graph algorithm animation system and its empirical evaluation as a student study tool. *Themes in Education*, **3**(1), 35–52.

Satratzemi, M., Dagdilelis, V., and Evangelidis, G. (2001) A system for program visualization and problem-solving path assessment of novice programmers. *ACM SIGCSE Bulletin*, **33**(3), 137–140.

Sleeman, D., Putnam, R. T., Baxter J., and Kuspa, L. (1987) An introductory Pascal class: A case study of students' errors. In *Teaching and Learning Computer Programming*, R. E. Mayer (ed.). Lawrence Erlbaum.

Snyder, L. (1999) Being fluent with information technology. National Research Council, USA.

Spohrer, J. G. and Soloway, E. (1986) Analysing the high frequency bugs in novice programs. In *First Workshop on Empirical Studies of Programmers*, Ablex Publishing Corp.

Spohrer, J. C., Soloway, E., and Pope, E. (1985) Where the bugs are. In *Proceedings of the Computer–Human Interaction Conference*.

Xynogalos, S. and Satratzemi, M. (2002) An integrated programming environment for teaching the object-oriented programming paradigm. In *Proceedings of the EurAsia-ICT 2002 Conference*, Shiraz, Iran. Springer, p. 2510.

Ziveldis, A. and Kappas, K. (2003) Some results from the teachers of informatics training. Under publication (in Greek).