Tackling barriers in the learning of computer programming

André Koscianski\textsuperscript{1} Elena Marieli Bini\textsuperscript{2}

\textsuperscript{1} PPGECT Postgraduate Program at UTFPR, Brazil, koscianski@utfpr.edu.br
\textsuperscript{2} Postgraduate at UTFPR-PPGECT, Brazil, professoraelena@yahoo.com.br

Abstract: The discipline of introduction to computer programming poses several difficulties to students and teachers. Success rates are low, students feel demotivated and there are cases of recurrent failure. Programming is an essential pre-requisite to other subjects in computer science, making the situation still worse. Moreover, several curriculum disciplines influence the programming contents, making it unlikely that a unified syllabus can be found. The problem is well known in the literature and, due to the multitude of aspects that affect it, currently there is no definitive solution. This article reports an on-going research, exploring some alternatives for the first programming discipline of a short graduation course. The methodology under study focuses on motivation and problem solution and pushes into the background aspects of less pedagogical interest in the selection of the programming languages and the problems presented to the class. We also introduce an innovative strategy, by using two programming languages. The first results are encouraging; the students got involved and there were good indications of authentic attempts to solve problems, replacing the more usual passive attitude.

Keywords: programming, teaching-learning, programming languages, ludic activities.

1. Introduction

Programming is a fundamental skill in computer science courses. Disciplines of introduction to computer programming may be found in several engineering degrees and are also proposed as optional activities on many high schools, or even earlier in primary education. For instance, the programming language Logo was created to explore logical thinking with children. In computer science degrees, the discipline of programming is an unavoidable pre-requisite to other subjects.

The development of programming skills strongly relies on problem-solving abilities. Students must learn textual and pictorial representations, get used to one
or more software tools and begin to develop their own algorithms. The abstract aspects of the subject contribute to the difficulty of the subject. Passing rates are low and many students feel demotivated. The problem is reported worldwide, not being bound to a particular culture or to unfamiliarity with the English language.

Handling abstractions is one of the most important barriers to the students. Indeed, the construction of mental models is a fundamental step on the development of programming skills. Visualization can be an effective scaffolding tool to reduce the abstraction effort; moving objects are more tangible than solutions to mathematical formula and make evident the dynamic character of a program. With the help of simple animations it is possible to create little games, adding a beneficial ludic character to the discipline.

This work reports an on-going study, where teaching strategies to postpone abstraction and emphasize motivation are used in introductory programming classes. We also tested the use of two programming languages, starting from an intuitive approach towards the solution of ‘real’ problems.

2. Overview of the Problem

Computer programs are generally represented as texts, with the exception of a few pictorial languages. Software is composed of instructions that are executed by a processor; these instructions are electrically represented as bits inside the memory. Special programs translate the textual, human-readable instructions, into a binary representation suited to a particular machine. This process is known as compilation.

The design of programming languages is affected by several factors, like easiness of translation and human comprehension and functionality provided. Programmers may write thousands of lines of code a month; as a consequence, conciseness is a major feature and most languages explore heavily the use of punctuation and operator signs. Unfortunately, this has a negative impact on readability and requires an extra attention effort. As an example, the following code fragment was intended to compute the sum \(0+1+...+n\):

```plaintext
sum = 0;
for (i = 0; i <= n; i++) ; // misplaced semicolon
sum = sum + i;
```

The lines above are syntactically valid in the languages C, C++ and Java, but the misplaced semicolon at the end of the second line prevents the code from working as intended. This error is hard to spot and illustrates a common difficulty faced by the students. Punctuation is, nonetheless, a minor concern in the whole process of implementing computer programs. A statement from Garner gives an accurate perspective of the problem [1]: “It is well known that learning to program is a difficult and frustrating process.”

The literature on the subject may be divided into two major axes. First, there are texts with an epistemological perspective that strives to trace the problem back
to cognitive and mental phenomena; we also place here the creation of frameworks to explain how a problem can be solved. The second axis has an empirical character and regroups studies of the effects of different languages and tools on the outcome of the discipline.

Teaching and learning aspects of computer programming have been studied even before the profession was born. Fundamental techniques for problem solution, like divide and conquer, have been incorporated in the first programming languages; subroutines were included in the language Plankalkül in 1945 [2]. The use of structure and hierarchy are constant characteristics of software design and programming; they help deal with the complex task of creating algorithms [3]. Stepwise refinement is a foremost technique, described decades ago [4].

The general problem of conceiving algorithms has drawn the attention of Polya, a mathematician who devised a series of steps to help this task [5]. The idea of partitioning can be frequently found; a possible subdivision is design, comprehension, composition, debugging and modification [6]. This type of classification can be used to spot students’ difficulties and serves as a means to structure instructional designs. Another path of investigation turns attention to the construction of knowledge and the mental models that could represent the functioning of a program [7, 8]. Recursive algorithms are treated as a class on its own, both from the perspective of programming and of learning [9, 10].

There is a huge diversity of languages and tools available for professional programming; only a few were created specifically for teaching. The most representative is Pascal; Modula-2 and Oberon were also developed with pedagogical objectives [11]. More recently, there has been a tendency to combine languages with computational environments dedicated to teach. Examples are Logo [12], Alice from Carnegie Mellon [13] and Scratch from the MIT [14, 30].

C++ and Java are very common initial languages; this is a result of their wide acceptance on market, despite their undesirable syntax complexity [15]. Visual Basic presents an uncluttered syntax and is widely used in practice, but paradoxically is not cited as a first language. Python also has characteristics that make it an excellent choice [19, 29]. Languages like Haskell and OCaml are rarely cited [16], despite studies suggesting that functional languages are easy to learn and have higher productivity [17, 18]. Some tools have been conceived for Portuguese language speakers, leading to a family of “portugol” languages in Brazil. The hypothesis at the origin of these tools is a supposed difficulty caused by a twenty-word English vocabulary. This claim has not been proved so far and the learning problems still persist.

Finally, the connections between learning theories and the discipline of computer programming remain largely unexplored; the main references are constructivism [20] and Papert’s “constructionism” [12]. The latter has particular importance due to the references to the use of computers. The theory of advanced organizers [21] may play a role in the instructional design, to orient the work with prerequisite skills as planning and abstracting. Approaches that concentrate on these abilities include metaphors [22] and micro-worlds [23]. Two interesting
examples in this category were mentioned: the tools Alice and Scratch. They use animated sprites and a drag-and-drop interface that avoids syntax errors.

3. Shaping a teaching approach

3.1 General aspects

Different from other subjects as calculus, computer programming can provide immediate, tangible responses. Software asks for inputs and show results, can record data in files or produce images. This feedback has high pedagogical interest, as it contributes to maintain the focus on the activity and helps self-assessments. It should facilitate the progression of the students, reducing their dependency on the teacher to evaluate each step of their work. Unfortunately, in practice two aspects reduce or completely prevent the use of immediate feedback.

First, despite the fact that a program can provide visible results, its internal logic is abstract. The distance between the static representation and its execution is known as “semantic gap” and causes problems even to experienced programmers.

Second, there is a reluctance of many instructors to employ the computer feedback. The learn-by-doing method is rarely used, because there is a belief that it fatally leads students to apply a strict trial-and-error approach:

“Programming teachers, being programmers and therefore formalists, are particularly prone to the ‘deductive fallacy’, the notion that there is a rational way in which knowledge can be laid out, through which students should be led step-by-step.” [24]

Papert has questioned this paradigm a few decades ago. He employed a learn-by-doing approach to teach programming to children, which has been successfully used by hundreds of teachers so far. The Logo environment is the foundation of Papert's work and has three characteristics that are particularly noticeable. It outputs drawings to the screen, enabling students to literally see the functioning of programs. Second, drawing is much more fun then creating algorithms to compute a sine with a Taylor series, or averages of lists of numbers; these are indeed usual tasks found in introductory courses. Finally, the syntax of Logo is minimalist. All the implementations follow this pattern, leaving no room for the weird syntax errors and semantic surprises that plague languages like Java and C++.

Logo is a particular case of the use of micro-worlds, existing several studies and educational tools in this category [13, 23, 25, 26]. They shift the focus from abstract problems to visualizable, controllable behaviors. Computer games can also be viewed as genuine examples of micro-words, offering the same learning support [28]. A somewhat related approach is to teach using examples, containing
characteristics and behaviors that students must modify [27]. This is easier then crafting a whole program starting from its specifications.

The above considerations suggest an approach where, in a first moment, the internalization of concepts and intuitive results should gain more emphasis than formal aspects. Students should comprehend the basic mechanics of computers before being exposed to abstract problems. Still in this sense, the choice of the tools used in an introductory course should accord more importance to pedagogical criteria, instead of acceptance in the market.

### 3.2 Survey with students

In order to proceed with the investigation, a survey was carried out. A questionnaire was prepared for introductory programming classes and applied at six institutions with geographical proximity to the university where the study was done. They were divided into three groups: high school, private universities and public universities. A total of 137 students with an average age of 18 answered the questions that were related to three main aspects:

- the balance between abstract and practical learning;
- the techniques learnt and employed to create programs;
- the difficulties faced on the whole process.

More than 70% of the students stated that classes took place in the laboratory in the beginning of the semester. While this should be an indicator of a learn-by-doing approach, in fact it is unlikely that the initial activities were really related to algorithm development. All the six institutions use Pascal and derivatives as the first programming language. This requires an introduction to the language syntax before it is actually used. By contrast, tools like Alice and Scratch can reduce the dependency on syntax to a minimum.

The use of structured natural language in classes was intense, according to 73% of the students. At the same time, more than 60% mentioned that never learnt nor used stepwise refinement; half of the answers showed a tendency to write code without a previous planning of the solution. This is in accordance with the argument of Dehnardi and Bornat that teachers strive to work with languages and formal approaches, pushing to the background techniques for actual problem solving [24]. Effectively, the use of a formal, logic-mathematical treatment should go on pair with the application of stepwise refinement or a technique like the one proposed by Polya. This aspect seems forgotten in several contemporary text books and in courses where a strong emphasis is placed on teaching commands instead of developing problem solving skills [31].

The third topic of the survey aimed to identify sources of difficulty experienced by the students. Syntax errors were classified as moderate or very difficult in 50% of the answers, while the comprehension of programming assignments received the same evaluation from 77% of the students. The abstract character of the
problems is likely to have an influence; printing multiplication tables and creating stock controls were the most mentioned exercises.

3.3 Outline of a proposal

Some of the factors identified as having higher incidence on the difficulty of the students are:

- the syntax of the programming language;
- the level of problem abstraction;
- the lack of motivation and sense of purpose.

There is a good agreement about the impact of programming languages on the learning process. For instance, the languages C and C++ are considered to be harder than Pascal, due to characteristics as weak typing and rich sets of operators. Basic and Python are easier than Pascal, but are not used in industry [16]. Since our objective is to prioritize the pedagogic aspect, Python is undoubtedly a good choice. The language uses much less punctuation than the C-lineage and automatically enforces the indentation of the code; as a result, block marks are unnecessary, text are less cluttered and better organized. Another strength of Python is the fact of being interpreted: this avoids the need to compile and provides feedback faster than C or Java.

If industry acceptance is an unavoidable curriculum requirement, a possibility is the use of two languages during the course. In an initial phase, basic programming tasks are introduced with the help of a simple programming language. Students will understand fundamental mechanisms by running their own, simple programs; this helps them to internalize concepts and create mental models. When a minimum skill is acquired, the class switches to a language specified by the school curriculum. At that point the students should already have grasp the basic knowledge on program creation; a programming language should be understood as a medium to express ideas and not as a tool to solve problems.

One of the authors has used this approach in two occasions. In a first attempt with Basic and Pascal and in the second, with an interpreter specially implemented. It avoided unnecessary punctuation and keywords and featured graphical capabilities. In both experiences, the transition between languages did not pose problems to the students.

Once the language has been chosen, the next difficulty is abstraction. We envisage two strategies to deal with it: emphasis on the learn-by-doing approach; and the use practical and motivating problems, like simple games.

Instead of starting the course with a theoretical view, actual computer programs may be presented very soon – possibly in the first day using a projector. Seeing an actual program sparks the curiosity and increases engagement. Initial examples should use only trivial assignments, calculations and PRINT and INPUT commands, and the students would be asked to participate on discussions to modifying the code. Details about program structure and data types are not
Tackling barriers in the learning of computer programming, WCC 2008 Congress

Important at this moment; they will be added later, by gradually introducing problems where these concepts play an important role.

A short example, this time in Python, can illustrate the idea. When the students have already learnt how to use variables, a code fragment to introduce loops could be presented as following:

```python
for i in range (10) :
    print i
```

When the program is executed, it shows the numbers 0 to 9. The functioning of the loop is discussed and students are invited to modify the code to print the range 0..19. When the correct answer is found, they are asked to modify the code to print the series in reverse order. This can be done this way:

```python
for i in range (20) :
    print 19-i
```

The problem illustrates how the students must construct a simple solution from elements already known. A similar task would be to print a range like 81..134; if the class get stuck, the teacher may present intermediary problems, like printing the ranges 81..83 or 81..91, until a correct answer is found in the class:

```python
for i in range (135-81) :
    print 81+i
```

This sequence of explanations quite often provokes exclamations like “but it was so simple!” It helps to encourage the students to keep trying and maintain the class highly focused. On the other hand, it requires a precise planning from the teacher, in order to ensure three pedagogic elements:

- presentation of feasible problems;
- gradual increase of difficulty, strictly controlled;
- a clear connection when new subjects are introduced.

Another great ingredient to promote motivation and to regulate the inclusion of abstraction are games. They require the handling of graphics, but this can be treated in a straightforward way by using a trivial tool: information hiding. In order to use it, the teacher only has to implement a simple library with two functions, capable of showing bitmap images obtained from the disk:

```python
show_image ("monster.bmp", x, y)   // loads and shows image
clear_image("monster.bmp", x, y)   // clear bounding box
```

In the case of Python, this programming environment can be created with an abstraction layer on top of the library PyGame. Next, the students receive simple assignments using these functions, like showing an image that follows keyboard commands. Once this is done, new challenges are presented, like adding random movements. Using this strategy, a high-school class of teenagers was able to write
variations of a shooting game in Pascal, with moving spaceships and projectiles. The best students manage to record scores in disk files.

Another possibility for handling graphics is the use of specialized tools; currently a test is being carried out with the tool Scratch. It goes one step further in replacing the pure code-writing approach: programs are created with a drag-and-drop interface, where graphical blocks represent loops, decisions and other commands, like the control of sprites and even playing music. The programs do not require compilation and can be immediately tested.

We also tested Scratch and the first reactions were very positive. The sole complain was the childish appearance of the sprites provided in the tool; once it was learnt how to change the images, the class got involved and started creating their first applications.

The next phase of the on-going research is to turn the accumulated experience into a systematic approach. A list of subjects and assignments will be defined, obeying a gradual increase of difficulty and providing opportunities for subject review. The combined use of multiple programming languages will be organized in a schedule, where the right moment for introducing each tool will be identified.

4. Conclusions

The discipline of introduction to computer programming presents a series of particularities that sharply distinguish it from other subjects. The essence of the course is problem solving; as a consequence, the development of this skill should be a central objective. However, several barriers complicate this task. Teaching and learning a skill is much harder than working with information primarily targeted to memorization. In programming classes, each assignment can represent a completely new problem that students are required to solve from the ground up. A wrong choice for the assignments, their sequencing and difficulty level can lead to irremediable frustration: many students give up and stop trying to solve problems, waiting for ready answers and examples.

The distance between the behavior of a computer and the textual representation of this behavior, known as the semantic gap, contributes for the abstract character of code and requires additional efforts from the students. Another problem comes from the fact that few programming languages were designed with pedagogic purposes. The tension between the teaching needs and the preparation of students for industry generally is won by the latter.

There are, nonetheless, several strategies that can ameliorate the situation. A shift of focus, from a pure formal treatment towards practical, fun activities, can be beneficial to the students and even make the teacher’s work more pleasant. In particular, abstract problems should be avoided during the first contact of the students with computer programming. Our first attempts in this sense used information hiding and the minimal examples strategy to implement simple
computer games. Currently we are testing the use of the programming tool Scratch, with initial positive results.

The use of simple languages during a short initial phase of the discipline prepares the class to meet complex requirements of a given syllabus. The authors have successfully used this approach; recommended languages for the first phase are Basic and Python. They have characteristics that correspond to pedagogic requirements, like reduced syntax complexity and rapid feedback.

Acknowledgment

The authors wish to thank the anonymous reviewers for their helpful comments.

References