

Programming fundamentals and human factors: an empirical study of three variables

Osvaldo Luis de Oliveira¹, Ana Maria Monteiro¹, Norton Trevisan Roman²

¹Faculty Campo Limpo Paulista – Campo Limpo Paulista, SP – Brazil

²University of São Paulo – São Paulo, SP – Brazil

osvaldo@faccamp.br, anammont@cc.faccamp.br, norton@usp.br

Abstract. In this study, we identify and experimentally investigate three important variables that are present in environments for the learning of programming fundamentals. These are the type of the problem source (concrete vs. abstract); the type of the programming language grammar (context-free vs. natural language-like); and the distance between the concepts in the problem source and the programming language primitives (close vs. distant). Results show that (i) there is no significant evidence to support any influence that the type of problem source or the type of programming language grammar may have in the learning of programming fundamentals; and (ii) languages whose primitives are close to the problem source concepts favor the learning of programming fundamentals when compared to languages whose primitives stray from these concepts. We understand that these results can be used to design better courses and learning material to improve students' performance in the learning of introductory programming.

1. Introduction

Computer Science and several technological fields rely on computers and the software that is used on them. The software developers must have programming skills that is why a student needs years to become a good programmer.

Learning and teaching programming is considered a difficult task because it requires a correct understanding of abstract concepts and the development of problem-solving strategies. For years, instructors have reported high failure rates in their courses [Devey and Carbone, 2011; Guzdial and Soloway, 2002], and research reveals that the dropout rates in the first two years of Computer Science programming courses are between 30% and 70% [García-Mateos and Fernández-Alemán, 2009; Moskal *et al.*, 2004].

Decades of research have been devoted to alleviating the problem of learning and teaching programming. Several methods and various methodologies have been proposed, which are based mainly on the global effectiveness of classroom experiences [Moskal, Lurie and Cooper, 2004]. Despite the large amount of research that has been conducted, there is currently no consensus on the most effective ways to learn programming. As such, beginning programming courses still have high dropout and failure rates [Mullins and Conlon, 2008].

With these considerations in mind, we begin our research to determine and elucidate some variables that potentially influence the learning of programming fundamentals, i.e., the learning of basic programming concepts such as sequences of sentences that express actions, conditions and repetitions. To do this, we have defined a set of **sources of problems** (or **domains of problems**) areas about which programming problems are proposed. A source of problems comprises a theme and involves elements and relationships between them. Algebra is an example of a source of problems that is commonly used in the learning of programming. In addition, we have developed some programming languages and compilers, along with **Integrated Programming Environments (IPEs)** comprising, among other things, a set of tools used to edit, compile, run and debug a program.

Such environments are necessary to investigate experimentally the influence, on the learning of programming fundamentals, of the following variables, which will be described in detail in Section 3.2:

- v1: Type of problem source: concrete or abstract;
- v2: Type of programming language grammar: traditional context-free or natural language like; and
- v3: Distance between concepts in the problem source and the programming language primitives: close or distant.

Within this setup, a **programming learning environment configuration** is a system comprising the following elements: students, teachers, a single source of problems, problems, one programming language and one IPE. For simplicity, considering the focus of this article, we will denote a programming learning environment configuration as a set of two elements {source of problems, programming language}, although all of the configurations bear the six types of elements mentioned above.

Finally, the **learning of programming fundamentals** refers to the introductory learning of the writing of sequences of sentences that express actions, conditions and repetitions to create programs that solve a problem.

The remainder of this article is organized as follows. Related work is described in Section 2. In Section 3, we structure the research, declaring hypotheses about the effects of the variables in the learning of programming fundamentals, along with the materials and methods used in the experiments. The results are described in Section 4 and are discussed in Section 5}. In Section 6, we present our conclusions.

2. Related Work

The importance of programming for the modern society and the difficulty of learning programming fundamentals have motivated the study of this theme from different perspectives by many research groups. The PPIG (Psychology of Programming Interest Group) brings together people with different backgrounds to explore common interests in the psychological aspects of programming [PPIG, 2014]. This group, which is active since 1987, organizes annual workshops, publishes regular newsletters, and hosts a discussion list. The SIGCSE (Special Interest Group on Computer Science Education

[SIGCSE, 2014]) and SIGITE (Special Interest Group for Information Technology Education [SIGITE, 2014]) are forums with general interests in computer education which often schedule exclusive sessions in their meetings to address this issue. Focusing on the design, implementation, and efficient use of programming languages, the SIGPLAN (Special Interest Group on Programming Languages [SIGPLAN, 2014]) has also investigated the difficulties in the learning of programming. Finally, the SIGCHI (Special Interest Group on Computer-Human Interaction [SIGCHI, 2014]) has dedicated efforts to understand the human factors that affect the interaction with computer programming systems.

Within this context, an environment for learning programming fundamentals is a system that traditionally involves (1) one or more sources of problems and (2) a programming language, in addition to students, teachers, and an environment for writing programs. This work aims to investigate a question about the source of problems, a question about programming language, and a question about the relationship between source of problems and programming language.

The question of the influence of the type of source of problems, concrete or abstract, is on the agenda of many working groups. The great motivation for this question lies in Piaget's theory of cognitive development [Piaget and Inhelder, 1972]. This theory suggests that child development moves from the concrete stage to the abstract stage. In the concrete stage, a child learns about tangible things, which are directly accessible to their visual, auditory, tactile, and kinesthetic senses. This stage occurs when the child is approximately 7 to 11 years old. Along the years, the child develops the ability to understand more abstract concepts, manipulate symbols, logically reason and generalize over things, moving to the abstract stage. The question then becomes whether Piaget's theory of cognitive development could be generalized to the learning of programming fundamentals for undergraduate students and, more specifically, if this learning should necessarily evolve from concrete to abstract problems? These are questions which are still in the need for answers.

Many studies about introductory programming have been proposed to explore concrete sources of problems: low-cost robot kits (*e.g.* [Summet et al., 2009; McWhorter and O'Connor, 2009]), graphical interfaces that simulate robot inhabited micro-worlds (*e.g.*, [Xinogalos et al., 2006; Pattis, 1995]), image processing (*e.g.*, [Wicentowski and Newhall, 2005]), computer networks (*e.g.*, [Murtagh, 2007]), geometrical drawing (*e.g.*, [Kordaki, 2010]), and 3D (*e.g.*, [Mullins and Conlon, 2008; Moskal et al., 2004}) and 2D (*e.g.*, [Resnick et al., 2009]) multimedia animation. In a systematic literature review covering 36 papers, Major *et al.* [Major et al., 2012] examine the effectiveness of using physical and simulated robots in the learning of introductory programming. On this matter, the Alice 3 programming environment was designed to suggest the movement from concrete to abstract: "the teacher can gradually lead students from the concrete context of animation to abstract data and structures in Java and a traditional context" [Dann and Cooper, 2009, pp. 29].

The difficulties imposed by the programming language used in introductory programming courses is another question that has occupied much space on the agenda of different research groups. With the premise that learning a context-free formal language represents a hurdle to the learning of programming fundamentals, many studies propose

tools and pedagogical strategies to facilitate this learning: interactive learning objects (*e.g.*, [Villalobos et al., 2009]), visualization tools for lecture demonstrations and course assignments (*e.g.*, [Kasurinen et al., 2008]), program representations in a flow-control diagram (flowchart) (*e.g.*, [Lavonen et al., 2003]), drag-and-drop tools to make it easier to write programs (*e.g.*, [Esteves, 2008; Klassen, 2006; Mullins and Conlon, 2008]), and suggestions to solve the problem in English (or native language) first (*e.g.*, [Fidge and Teague, 2009; Kaplan, 2010]).

Another line of research proposes to investigate how programming languages might be designed to facilitate the learning of programming fundamentals. A branch of this line of research investigates how to close the gap between programming languages and natural language. Empirical studies (*e.g.*, [Chen et al., 2007; Simon et al., 2006; Lewandowski et al., 2007]) show that novices exhibit fairly advanced problem-solving skills when they express the solution in English; however, programming languages create difficulties for them to express these same skills. HCI (Human Computer Interaction) knowledge has been used to develop new programming and debugging tools which enable people to express their ideas in the same way they think about them [Myers et al., 2004]. Examples of systems developed for experimenting with natural language programming include the pioneer MIRFAC [Gawlik, 1963], which was specifically designed for the mathematical formulas domain, NL [Biermann et al., 1983], which was designed for the tables and matrices domain, and PEGASUS [Knöll and Mezini, 2006], which is an active project that facilitates programming in German or English in several domains.

Although both previously mentioned questions have taken a good deal of space in the research agenda of many groups, the question about the influence of the distance between concepts in the source of problems and the programming language primitives is new. This study differs from existing research in this sense, but mainly because, methodologically, it proposes to separately investigate the influence of each variable, described in the previous section, while holding all other variables constant. Languages, compilers, and integrated programming environments were specially developed for the set of experiments conducted. A further particularity that differentiates this study is that it focuses on learning sequences of sentences that express actions, conditions, and repetitions.

3. Material and Methods

In our experiments, we investigate the isolated influence of the variables v_1 , v_2 and v_3 in the learning of programming fundamentals by students who have no previous background on this subject. The tested hypotheses were:

- Hypothesis 1: A concrete problem source is better for the learning of programming fundamentals than an abstract one;
- Hypothesis 2: Programming languages whose grammars are close to the natural language spoken by the student are better for programming fundamentals learning than those with traditional grammars; and

- Hypothesis 3: Languages whose primitives are close to the concepts in the problem source favor the learning of programming fundamentals when compared to languages whose primitives stray from those concepts.

3.1. Programming Learning Environment Configurations

To test the setup hypotheses, we designed four programming learning environment configurations:

- World of the Robots and MRt Language;
- World of the Robots and MRp Language;
- Small Algebra and Pascalish Language;
- Academic Record Management and Pascalish Language.

Configurations 1 and 2 use the World of the Robots problem source, being different only in their programming language. Whereas MRt is a language that is a language defined by a context-free grammar, MRp bases its sentences on Portuguese, therefore a natural language. Configurations 3 and 4, on the other hand, work on the Small Algebra and Academic Record management problem sources, respectively, both using Pascalish, a language based on Pascal (*i.e.*, a language defined by a traditional context-free grammar), with statements translated to Portuguese.

Amongst all possible combinations for the variables under examination, we constrained our research to this set of four basic configurations to focus on answering the questions elicited by the tested hypotheses.

3.1.1. World of the Robots and MRt Language

Following Pattis [Pattis, 1995], the World of the Robots comprises a rectangular board (representing the “world”), robots, walls and disks. This setup leads to create problems related to the movement and manipulation of objects (disks) in the board, which can sometimes present some barriers (depicted as walls) to the robot. Figure 1 presents a screenshot of the World of the Robots. Its IDE includes a graphical editor (which is used to build scenarios in the World of the Robots, as shown in the figure), a text editor (for coding the program), compilers and debuggers for the MRt and MRp languages (which are described in the next sections).

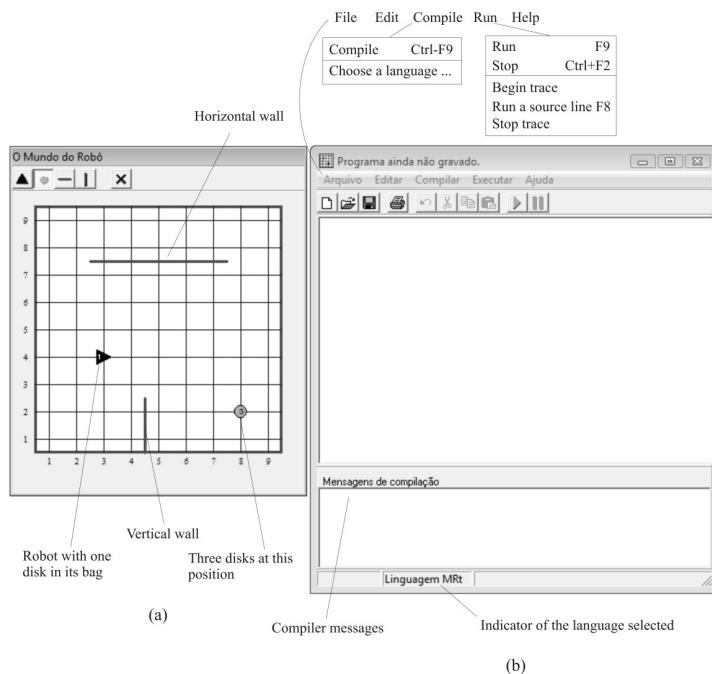


Figura 1. The IPE of the {World of the Robots, MRt} programming learning environment

Through the selection of buttons in the graphical editor's tool bar, the user can create and delete robots, disks, vertical and horizontal walls. In response to the request for executing a program, the programming environment shows an animation of the programmed movements. It is also possible to run programs in debug mode, line by line, so that their execution can be traced and compared to the corresponding animation in the World of the Robots.

Because MRt was designed to accommodate traditional context-free grammar structures under the imperative paradigm (such as those found in programming languages like Pascal and C), it allows for the following¹:

- Sentences such as *MoveForward(r)*, *TurnLeft(r)*, *TakeDisk(r)*, and *DropDisk(r)*, which will have the robot *r*, respectively, move forward, turn left (90°) and take or drop a disk (at the robot's current position);
- Boolean expressions that can be built from boolean primitives such as *FrontClear(r)*, *OnDisk(r)*, *ThereAreDisksInBag(r)*, *HeadsNorth(r)*, *HeadsSouth(r)*, *HeadsEast(r)* and *HeadsWest(r)*, with the operators *and*, *or* and *not*. These primitives evaluate to true, respectively, if there is no wall before *r* (the robot), if *r* lies above a disk, if there is at least one disk in *r*'s bag, and if *r* is heading (*i.e.*, points toward) north, south, east or west;
- Conditional sentences, such as *if FrontClear(r) then MoveForward(r) else TurnLeft(r)*, that test whether the space before *r* is clear and, if so, make the robot move forward or, otherwise, turn left;

¹ Grammars developed in this study use terms in Portuguese. Aiming to facilitate the reading of this article, we translated into English the sentences derived from these grammars.

- Repetition sentences, such as *while FrontClear(r) and OnDisk(r) do TakeDisk(r)*, which repeats the instruction for the robot *r* to collect a disk if there is any left at its position and the space ahead is clear;
- Sets of sentences, grouped together by *begin* and *end* delimiters;
- An identification to the program, with the reserved word *program*; and
- The definition of variables (robots and disks), as in *uses list of robot names: Robot*.

Figure 2 illustrates an initial scenario of the World of the Robots (left) along with a complete program (right) that was written in MRt to solve the problem of making a robot *r* find a disk that is adjacent to a wall.

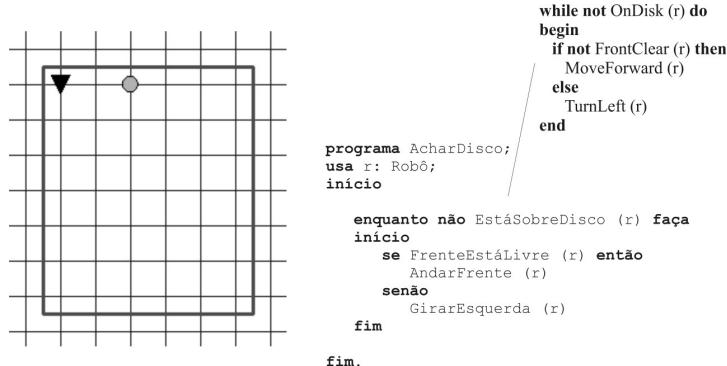


Figure 2. Example of a complete program written in MRt.

3.1.2. World of the Robots and MRp Language

This configuration differs from the previous one in that MRt is replaced by MRp as the adopted programming language. The two languages differ in the way that statements are written: whereas MRt provides a traditional context-free grammar, MRp is based in Portuguese, a natural language. Figure 3 shows some sample sentences generated by the MRp grammar.

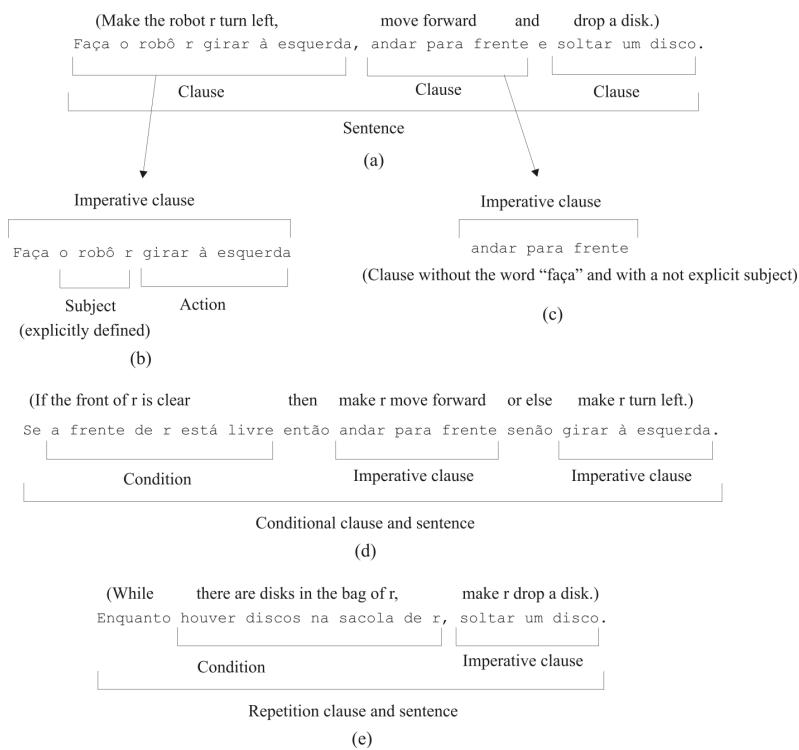


Figure 3. Sample sentences and clauses in MR

The body of a program written in MRp comprises a sequence of zero or more sentences, each ending with a dot. Each sentence comprises one or more clauses, which are separated by commas or by the conjunction *and*, as shown in Figure 3-a. Within MRp, the clauses can be either imperative (Figures 3-b and c), which may or may not begin with the word *make*; conditional (Figure 3-d), which are characterized by the words *if*, *then* and, optionally, *else*; or repetitious (Figure 3-e), which always begins with the word *while*. Additionally, clauses always have a subject, which can be made explicit (as in Figure 3-b) or not (as in Figure 3-c), in which case the subject corresponds to that of the previous clause (in Figure 3-c, the subject is therefore *r*).

Conditions can be plain statements, such as *the robot r is heading north*, which semantically evaluates to true if *r* is pointing north, or more complex structures, which are built from such statements joined with the words *or*, *and*, or *not*. Figure 4 shows a sample program, written in MRp, and its corresponding “world”, to solve the same problem presented in Figure 2. To improve readability, the program is shown in MRp along with its translation to English

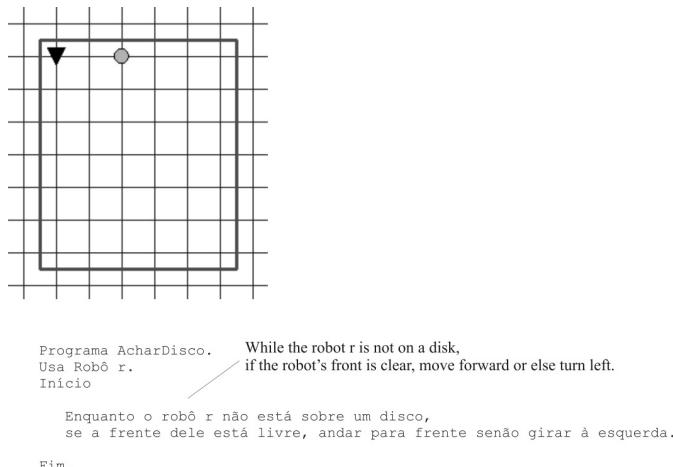


Figure 4. Example of a complete program written in MRp.

3.1.3. Small Algebra and Pascalish Language

Traditional environments for the learning of programming fundamentals frequently take Algebra, with its sets, functions, expressions and equations, as their source of problems. Languages under the imperative paradigm, such as Pascal or C, for example, are commonly used to implement programs that describe solutions to problems in this source. For the purposes of this research, we used a subset of Algebra (called Small Algebra) as our problem source, along with a subset of Pascal, with statements written in Portuguese, which we call Pascalish, as the programming language to be used when solving these problems.

Small Algebra comprehends:

- Sets of integers and floats;
- Constants and variables (also integers and floats); and
- Algebraic expressions, which involve constants, variables and arithmetic (+,\$ \$-, *, /, mod), boolean (or, and, not) and relational (<, <=, =, >, >=, >) operators.

Pascalish was designed so that its primitives directly reference elements in the Small Algebra problem source, thereby allowing for constants, variables and expressions to be used in a way that resembles regular algebraic statements. An example of a program written in Pascalish is shown in Figure 5. In this figure, the problem to be solved is that of calculating the amount of even and odd numbers in a list of positive integers, taken one by one, and whose stop condition is the reading of a negative number.

```

programa Soma;
var somaPar, somaImpar, num: inteiro;
inicio
    somaPar ← 0; somaImpar ← 0;
    leia (num);
    enquanto num ≥ 0 faça
        inicio
            se num RESTO 2 = 0 então
                somaPar ← somaPar + num
            senão
                somaImpar ← somaImpar + num;
            fim;
            leia (num)
        fim;
        escreva ('Soma dos pares = ', somaPar);     write ('Sum of even numbers = ', somaPar);
        escreva ('Soma dos impares = ', somaImpar) write ('Sum of odd numbers = ', somaImpar);
fim.

```

Figure 5. Example of a complete program written in Pascalish

3.1.4. Academic Record Management and Pascalish Language

In this programming learning environment configuration, the source of problems is an adaptation of the academic record management system used in a university in Brazil. As such, it involves the following concepts:

- Test types (*i.e.*, either short-term, long-term or a general test);
- Individual scores (along with their mean values), obtained by the students in each test;
- Mean scores obtained by the students in a specific course, considering the different test types; and
- Class attendance frequencies, obtained by students in a specific course.

The problems in this learning environment involve calculations such as averages and frequencies and approvals check. The language used in this configuration is Pascalish.

3.2. Variables

Table 1 summarizes the investigated variables, their possible values and examples. Each variable will be treated in detail in what follows.

Table 1. Investigated variables

<i>Variable / Description</i>	<i>Possible values</i>	<i>Examples</i>
v_1 Type of source of problems	Concrete	1. World of the Robots 2. Academic Record Management
	Abstract	1. Algebra 2. Arithmetic
v_2 Type of language grammar	Traditional (context-free grammar)	1. Pascal, Pascalish 2. M _{Rt}
	Natural language like	1. M _{Rp}
v_3 Distance between concepts of the source of problems and the programming language primitives	Close	1. World of the Robots and M _{Rt} 2. Small Algebra and Pascalish
	Distant	1. World of the Robots and Pascal 2. Academic Record Management and Pascalish

3.2.1. Variable v_1 – the Type of Source of Problems

Problem sources can be concrete or abstract. In this article, we consider problem source to be **abstract** whenever its elements can be used to refer either to a set of objects and phenomena in the world or to another set. As such, elements in an abstract problem source are ideas and abstractions of objects and phenomena in the world. Conversely, a **concrete** source of problems is built from elements that keep a direct relation to specific sets of existing objects and phenomena. Elements in a concrete source of problems are, thus, specific instances of these objects and phenomena.

The World of the Robots is an example of a concrete problem source; another is Academic Record Management. The elements of these sources map directly to their counterparts in the real world. In contrast, Algebra and Arithmetic are examples of abstract sources of problems whose elements are generalizations of real-world concepts and operations (*e.g.*, the concept of an algebraic variable can be used to model both the temperature of thermodynamic systems and currency in market simulations). In this research, v_1 may take the values of “Abstract” or “Concrete”, depending on whether the source of problems can be characterized as abstract or concrete, respectively.

3.2.2. Variable v_2 – the Type of Language Grammar

Traditional programming language grammars, such as Pascal’s and C’s, differ to a great extent from those of natural languages. Hence, one might suspect that programming languages whose structures are closer to a natural language could facilitate the learning process by native speakers of the language, given the assumed reduction in the cognitive load during the learning of the grammar.

We account for this disparity by assigning to the variable v_2 the value “Traditional” whenever the learning environment has a programming language whose grammar belongs to the context-free class, according to Chomsky’s hierarchy [Hopcroft et al., 2006], which is the case of Pascalish and MRt. Alternatively, by assigning it the value “Natural language like”, we imply that the programming language grammar lies much closer to that of a natural language when compared to its “Traditional” counterpart. An example of such a language is MRp.

3.2.3. Variable v_3 – the Distance between Concepts in the Source of Problems and Programming Language Primitives

One might conjecture that if the semantics and grammar of a programming language match those of the source of problems, *i.e.* if the primitives of the language faithfully describe the concepts in the source, then the effort that is spent in writing programs might be reduced because strategies for addressing the elements of the source of problems will be more directly expressed by the structures of the language. This phenomenon would not occur, for example, in programming languages whose primitives stray from the concepts presented in the source of problems.

To test this assumption, we assign to the variable v_3 two discrete values, namely, “Close”, which indicates that the concepts in the source can be directly mapped to primitives in the programming language, and “Distant”, which is applicable whenever there is a need to model, translate or simulate these concepts using the language’s primitives. In this research, MRt is an example of a Close-type language for the World

of the Robots source of problems because its primitives, such as *MoveForward* and *TurnLeft*, are directly related to the robot's movements. In contrast, the concepts of World of the Robots would have to be translated into variables, constants and data types of a language like Pascalish, for example. In this case, we take Pascalish to be a Distant-type language for the World of the Robots. Other examples include that Pascalish is a Close-type language for the source of problems Small Algebra, being at the same time a Distant-type language for the Academic Record Management.

3.3. Participants

Forty-four first-year undergraduate students of a university in Brazil volunteered for the experiment. Of these, four did not show up for the experiment (*i.e.* a 9.1 % dropout rate). Of the remaining 40 participants, 24 (60 %) were male, whereas 16 (40 %) were female, with 22 (55 %) coming from the exact sciences and 18 (45 %) from the social sciences. The ages ranged from 17 to 33 years old, with 21 as the mean and 19 as the median. Additionally, none of the participants had previous knowledge of programming, as determined by a questionnaire.

3.4. Experimental Setup

We set up four different experimental conditions, each corresponding to a different programming learning environment configuration, shown in Table 2.

Table 2. Learning environment configurations and variables used in each experimental condition.

Group	Configuration {source of problems, programming language }	v_1	v_2	v_3
I	{ World of the Robots, MRT }	Concrete	Traditional	Close
II	{ Small Algebra, Pascalish }	Abstract	Traditional	Close
III	{ World of the Robots, MRT }	Concrete	Natural language like	Close
IV	{ Academic Record Management, Pascalish }	Concrete	Traditional	Distant

The initial set of participants (*i.e.* all 44) was randomly distributed among the experimental conditions. Because four of them dropped out by the time that the experiment was performed, the number of participants was not the same across the experimental conditions (the distribution was 11 participants for conditions I, II and III and 7 for condition IV). Due to time and resource limitations, the experiment was conducted in two different dates, separated by experimental condition. Hence, groups III and IV undertook the experiment one week after groups I and II.

For each of the experimental conditions, the experiment was split into two phases, designed to take place during the morning (phase 1) and afternoon (phase 2) periods of the same day, with a one and a half hour break for lunch between them. Phase 1 constituted 3 hours and 30 minutes classes, with one class per experimental condition and with a 15 minute break halfway through. During the classes, students were

introduced to the elements of their experimental condition's source problem, along with concepts such as sequence of sentences and sentences of the imperative, conditional and repetitive type, tailored to each condition's programming language. Before the 15 minute break, participants took part in an exercise, in which they had to address these concepts, applying them to the source of problems they were using. During this task, the teachers interacted with the participants on an individual basis. At the end, a possible solution was developed and was shown to all the participants.

The elements of each source of problems, along with sample problems and programs presented in Phase 1, are shown in Table 3. It is worth mentioning that, before taking the class, each participant was given a text that described the concepts to be learned in that class, so that they did not have to worry about taking notes on the entire lecture; instead, they only wrote down complementary notes as they felt necessary.

At phase 2, participants had up to two hours to answer a four-question evaluation test. Each question presented a problem and asked the participant to write a program to solve it. Tests were carefully elaborated so that questions would be equivalent across the experimental conditions, with regard to the concepts and skills necessary to solve the problems. Hence, for each experimental condition:

- Question 1 evaluated the use of sequences of imperative sentences;
- Question 2 evaluated the use of conditional sentences;
- Question 3 evaluated the use of repetitive sentences
- Question 4 evaluated the use of imperative, conditional and repetitive sentences altogether.

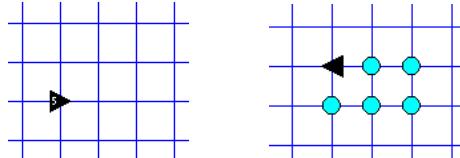
Table 3. Elements of the source of problems and program samples presented at phase 1 to each group

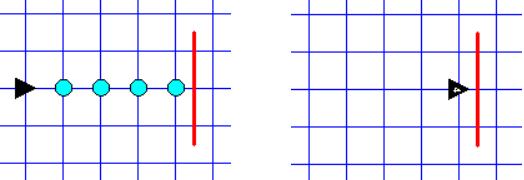
<i>Group</i>	<i>Elements of the source of problems</i>	<i>Sample problem</i>	<i>Sample program</i>
I	Robots, walls and disks.	Write a program to make the robot <i>r</i> turn left and take disks as many times as the quantity of disks that are placed under its position.	programa Girar; usa r: Robô; início enquanto EstáSobreDisco (r) faz início GirarEsquerda (r); PegarDisco (r) fim fim.
II	Integer and real variables.	Write a program to sum the first <i>n</i> positive integers.	programa SomaInteiros; var n, i, soma: inteiro; início Leia (n); i := 1; soma := 0; enquanto i <= n faz início soma := soma + i; i := i + 1 fim; Escrever (soma) fim.

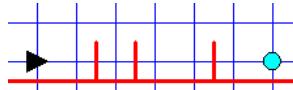
III	Robots, walls and disks.	Write a program to make the robot <i>r</i> turn left and take disks as many times as the quantity of disks that are placed under its position.	programa Girar. Usa Robô <i>r</i> . início Enquanto <i>r</i> estiver sobre um disco, faça ele girar à esquerda e pegar um disco. fim.
IV	Scores, means, frequencies and test types.	Write a program that, for each student in a discipline, reads the number of classes that he attended and outputs the student's frequency. First, the program must read the number of students and classes in a discipline.	programa Frequências. var aulas, presenças, n: inteiro; frequência: real; início Leia (aulas); Leia (n); i := 1; enquanto i <= n faça início Leia (presenças); frequência := 100*(presenças/aulas); Escreva (frequência) fim fim.

The questions presented in each experimental condition can be found in Table 4. The answers were separately analyzed by two of the authors, who assigned them a score from 0 to 5. The evaluators discussed disagreements on the scores until agreement was reached.

Table 4. Questions presented to each group at Phase 2

<i>Question</i>	<i>What was evaluated?</i>	<i>Could be answered with</i>	<i>Question description</i>
1	The use of a sequence of imperative sentences.	A sequence of imperative sentences.	<p>Groups I and III</p> <p>Write a program to signal a path with disks. The left figure presents an initial configuration in the World of the Robots and the right figure the final configuration that must be reached after executing the program.</p>  <p>Group II</p> <p>Write a program that reads the number of sides in a regular polygon, along with each side's length, and calculates its perimeter.</p> <p>Group IV</p> <p>Write a program to calculate the overall mean M of a student's scores from the mean in short-term (C), long-term (L) and general (G) tests. (At this point, an extra text reminded the student that $M = 0.1 C + 0.7 L + 0.2 G$).</p>
2	The use of Conditional	Conditional	Groups I and III

	conditional sentences.	sentences, nested or not.	<p>Write a program that makes the robot <i>r</i> point north. You do not know the initial configuration of the robot in the World, in other words, you do not know whether the robot is currently pointing north, south, east or west.</p> <p>Group II</p> <p>Given a second degree equation $Ax^2 + Bx + C = 0$, it has 1, 2 or 0 real roots depending on whether the result of $B^2 - 4AC$ is zero, positive or negative, respectively. Write down a program that reads the values A, B and C and informs how many roots the equation has.</p> <p>Group IV</p> <p>Write a program that takes as input the mean score of a student, the number of classes that he missed and the number of classes presented in the discipline. The program should inform whether the student has passed the exam, failed to attend the classes, or must undertake the final exam (an extra text describes the adopted criteria for approval).</p>
3	The use of repetition sentences.	A repetition of imperative sentences.	<p>Groups I and III</p> <p>Write a program to make the robot <i>r</i> take all disks in a line situated before it until it reaches a wall. The figures show a possible starting (left) and final (right) condition. These figures are only an example. Your program must work for lines with an undetermined number of disks.</p>  <p>Group II</p> <p>Write a program to calculate the sum of the <i>n</i> first even numbers that are higher or equal to 2. Hence, for example, if <i>n</i> is 5, then the program must calculate the sum of 2, 4, 6, 8 and 10 (2+4+6+8+10).</p> <p>Group IV</p> <p>Given 3 scores for the long-term tests of each of the <i>n</i> students in an Introduction to Philosophy course, write a program to calculate and inform the mean score of each student in the long-term tests.</p>
4	The combined use of imperative, conditional and repetition	A repetition of a condition with imperative sentences.	<p>Groups I and III</p> <p>Write a program to make the robot <i>r</i> jump over barriers such as those in the figure until it finds a disk. Again, this figure is merely illustrative. Your</p>

	<p>sentences.</p>	<p>program must address an undetermined number of barriers with unknown positions. The barriers are always equal in size to those indicated in the figure.</p> 
		<p>Group II</p> <p>Write a program that reads a number of positive integers (until the input number is -1) and calculates the number of odd and even numbers that were read. The number -1 must not be accounted for in the result.</p>
		<p>Group IV</p> <p>Write a program to read the students' final mean score and to inform the number of students whose mean is under 5, along with the number of students with a mean of 5 or more. The end of the input is indicated by a -1 score. This score must not be accounted for when calculating the result.</p>

4. Results

Table 5 shows the test results for each group. Groups I and II have the same value for variables v_2 and v_3 . The difference between them is the value of the variable v_1 , *i.e.* the source of problems. While Group I used a concrete source of problems, Group II dealt with an abstract one. The mean obtained by Group I (3.98) is higher than that obtained by the Group II (3.08). However, this result is not statistically significant ($W = 83.5$, $p = 0.14$)². As such, Hypothesis 1 could not be confirmed (*cf.* [Oliveira et al., 2011]).

Table 5. The results for all four experimental conditions

Group	Learning environment configuration { source of problems, programming language }	Values of the variables (v_1 , v_2 , v_3)	Mean 0 to 5	SD
I	{ World of the Robots, MRt }	(Concrete, Traditional, Close)	3,98	0.82
II	{ Small Algebra, Pascalish }	(Abstract, Traditional, Close)	3,08	1.35
III	{ World of the Robots, MRt }	(Concrete, Natural language like, Close)	3,91	0.79
IV	{ Academic Record Management, Pascalish }	(Concrete, Traditional, Distant)	1,93	1.08

² Since the data seem not to follow a normal distribution, we decided to use Wilcoxon-Mann-Whitney non-parametric test.

The experiments with Groups I and III differ only by the value of the variable v_2 , *i.e.* the type of language grammar used. While Group I used a traditional context-free grammar, Group III used a natural language like grammar. In this case, both means were almost identical (3.98 and 3.91, respectively), which, thereby, does not confirm Hypothesis 2 either ($W = 67.5$, $p = 0.67$).

Finally, the value of the variable v_2 is the only difference between Groups I and IV. While Group I used a language with primitives close to the problem source concepts, Group IV used a Distant-type language. We noticed a better performance by the first group. This result was statistically significant ($W = 73$, $p = 0.002$), which confirms Hypothesis 3.

5. Discussion

Our results show, for the studied population, that:

- There is no significant evidence to support that the type of source of problems (either concrete or abstract), or the type of language grammar (either traditional or natural language like), affect the learning of programming fundamentals; and
- Languages whose primitives are close to the problem source concepts favor the learning of programming fundamentals when compared to languages whose primitives stray from these concepts.

The question of the type of problem source (whether concrete or abstract) has occupied many research agendas. As an alternative to an abstract source of problems, such as Algebra, many educators and researchers proposed a concrete one for introductory programming course. However, the experiences reported are contradictory regarding the effectiveness of using a concrete source of problems in the learning of programming fundamentals.

Take Alice, for example, which is an environment for learning programming fundamentals [Dann and Cooper, 2009; Moskal et al., 2004; Mullins and Conlon, 2008] that has established, as one of its design principles, a concrete source of problems through the animation of objects and 3-D characters. While Moskal *et al.* (2004) reported improvements in student performance; Cliburn (2008) discourages the use of Alice in introductory courses.

Robot kits or computer-simulated robots also have been used to create a concrete source of problems for an introduction to programming. Here too, reported experiments are divergent. While Xinogalos *et al.* (2006) evaluate their system as an effective tool for teaching novice programmers; others report that these systems have little or no effect whatsoever on motivation [McWhorter and O'Connor, 2009] and students' performance [Summet et al., 2009].

We understand that our results are consistent with the divergences that are highlighted in the literature. Because we observed no evidence that the type of source of problems affects the learning of programming fundamentals, it is acceptable that different practical experiences and research have led to contradictory results. We conjecture that other variables have led sometimes to positive experiences and, at other times, to negative ones, in relation to students' performances.

The learning of a traditional context-free programming language grammar also has occupied research agendas. Fidge and Teague (2009) conjecture that the effort to learn a traditional programming language drives the student's focus away from the logic that defines the solution to the problems. Thus, some studies try to identify what language structures are natural for beginners (*e.g.* [Pane et al., 2001]), while other aim at facilitating the assimilation of the programming language grammar through Interactive Learning Objects (*e.g.* [Villalobos et al., 2009]), animations of program execution (*e.g.* [Hundhausen, 2002; Levy et al., 2003]), use of visual programming languages (*e.g.* [Navarro-Prieto and nas, 2001; Lavonen et al., 2003]), use of dragging-and-dropping blocks of sentences (*e.g.* [Resnick et al., 2009]), pair programming (*e.g.* [Hanks, 2008]) and even the use of auditory cues to enhance program comprehension (*e.g.* [Stefik et al., 2011]).

Our results show that the use of natural language for the learning of programming fundamentals is an alternative, no better or worse, to the use of programming languages based on traditional context-free grammars. This alternative is attractive because it means that is not necessary to learn a traditional programming language to learn programming fundamentals, and thus, it brings comfort to the learner and the teacher. This result does not contradict or disqualify the work that has focused on facilitating the learning of a traditional programming language. At some time, the student should be exposed to a traditional context-free programming language, but this exposure can be facilitated if the student is already aware of the programming fundamentals.

The relevance and complexity of learning programming fundamentals has motivated research on a wide variety of phenomena that are associated with this subject. Even so, the literature has not reported practices or research findings on the effect of the relationship between the distance of the primitives of programming languages and the concepts in the source of problems. We believe that this work is the first to make this effect explicit.

6. Conclusions

Traditionally, research on the learning of programming fundamentals is focused on investigating the global effectiveness of learning environments. This work differs from traditional research mainly because it makes the identification and isolated investigation of variables that are present in environments for learning programming fundamentals. We accomplished this goal for three variables. In future studies, other important variables can be investigated such as, for example: (1) the students' prior knowledge about a source of problems, (2) the motivation of students to solve problems of a specific source, (3) the usefulness of an integrated programming environment for writing, grammatical checking, execution and debugging of a program, and (4) gender.

Our results about the type of source of problems, the type of the programming language grammar and the relationships between the primitives of the programming language and the concepts present in the source of problems can be directly used by educators when designing educational environments, tools and instructional material for learning programming fundamentals.

We should note, too, that experiments that involve a larger population are important to ensure the generality of our results.

References

- Biermann, A. W., Ballard, B. W., Sigmon, A. H. (1983) An experimental study of natural language programming, *International Journal of Man-Machine Studies* 18, p. 71–87.
- Chen, T., Xavier, G. L., McCartney, R., Sanders, K., Simon, B. (2007) Commonsense Computing: using student sorting abilities to improve instruction, In: SIGCSE'2007: Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education, ACM Press, New York, NY, USA, p. 276-280.
- Cliburn, D. C. (2008). Student opinions of Alice in CS1, In: FIE'2008: Proceedings of the 38th ASEE/IEEE Frontiers in Education Conference, IEEE Computer Society Press, Los Alamitos, CA, USA, p. T3B1–T3B6.
- Dann, W., Cooper, S. (2009) Alice 3: concrete to abstract, *Communications of the ACM* 52(8), p. 27–29.
- Devey, A., Carbone, A. (2011) Helping first year novice programming students PASS, In: ACE'2011: Proceedings of the Thirteenth Australasian Computing Education Conference, Australian Computer Society, Inc, Perth, Australia, p. 135-144.
- Esteves, M. (2008) Contextualization of programming learning: a virtual environment study, In: FIE'2008 Frontiers in Education Conference, 22-25 October 2008, Proceedings of the 38th ASEE/IEEE Frontiers in Education Conference, New York: IEEE Publisher, p. F2A17-F2A22.
- Fidge, C., Teague, D. (2009) Losing their marbles: syntax-free programming for assessing problem-solving skills, In: ACE'2009: Proceedings of the 11th Australasian Computing Education Conference, Australian Computer Society, Inc., Wellington, New Zealand, p. 75–82.
- García-Mateos, G., Fernández-Alemán, J. L. (2009) A course on algorithms and data structures using on-line judging, *ACM SIGCSE Bulletin* 41(3), p. 45–49.
- Gawlik, H. J., 1963. MIRFAC: A compiler based on standard mathematical notation and plain english. *Communications of the ACM* 6(9), 545–547.
- Guzdial, M., Soloway, E. (2002) Teaching the Nintendo generation to program, *Communications of the ACM* 45(4), p. 17–21.
- Hanks, B. (2008) Empirical evaluation of distributed pair programming, *International Journal of Human-Computer Studies* 66, p. 530–544.
- Hopcroft, J. E., Motwani, R., Ullman, J. D. Introduction to automata theory, languages, and computation, 3rd ed. Addison-Wesley Longman Publishing, 2006.
- Hundhausen, C. D. (2002) Integrating algorithm visualization technology into an undergraduate algorithms course: ethnographic studies of a social constructivist approach, *Computers & Education* 39 (3), p. 237–260.

- Kaplan, R. M. (2010) Teaching novice programmers programming wisdom, In: PPIG'2010: Proceedings of the 20th Workshop of Psychology of Programming Interest Group, Madrid, Spain, p. 1–8.
- Kasurinen, J., Purmonen, M., Nikula, U. (2008) A study of visualization in introductory programming, In: PPIG'2008: Proceedings of the 22th Workshop of Psychology of Programming Interest Group, Lancaster, UK, p. 1–14.
- Klassen, M. (2006) Visual approach for teaching programming concepts, In: ICEE'2006: Proceedings of the 9th International Conference on Engineering Education, p. TIA 1– TIA 6.
- Knöll, R., Mezini, M. (2006) Pegasus – First steps toward a naturalistic programming language, In: OOPSLA'2006: Proceedings of the 21st Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Application, ACM Press, New York, NY, USA, p. 542–559.
- Kordaki, M. (2010) A drawing and multi-representational computer environment for beginners' learning of programming using C: design and pilot formative evaluation, *Computers & Education*, 54(1), p. 69–87.
- Lavonen, J. M., Meisalo, V. P., Lattu, M., Sutinen, E. (2003) Concretizing the programming task: a case study in a secondary school, *Computers & Education* 40(2), p. 115–135.
- Levy, R. B., Ben-Aria, M., Uronen, P. A. (2003) The Jeliot 2000 program animation system, *Computers & Education* 40(1), p. 1–15.
- Lewandowski, G., Bouvier D, J., McCartney, R., Sanders, K., Simon B. (2007) Commonsense computing (episode 3): concurrency and concert tickets. In: Proceedings of the Third International Workshop on Computing Education Research (ICER 2007). Atlanta, GA, USA, p. 133–144.
- Major, L., Kyriacou, T., Brereton, P. (2012) Teaching novices programming using a robot simulator: Case Study Protocol, In: PPIG'2012: Proceedings of the 24th Workshop of Psychology of Programming Interest Group, London, UK, p. 1–12.
- McWhorter, W., O'Connor, B. (2009) Do LEGO® Mindstorms® motivate students in CS1?, In: SIGCSE'2009: Proceedings of the 40th ACM Technical Symposium on Computer Science Education, ACM Press, New York, NY, USA, p. 438–442.
- Moskal, B., Lurie, D., Cooper, S. (2004) Evaluating the effectiveness of a new instructional approach, In: SIGCSE'2004: Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education, ACM Press, New York, NY, USA, p. 75–79.
- Mullins, P. M., Conlon, M. (2008) Engaging students in programming fundamentals using Alice 2.0, In: SIGITE'2008: Proceedings of the 9th ACM SIGITE Conference on Information Technology Education, ACM Press, New York, NY, USA, p. 81–88.
- Murtagh, T. P. (2007) Weaving CS into CS1: A doubly depth first approach, In: SIGCSE'2007: Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education, ACM Press, New York, NY, USA, p. 336–340.

- Myers, B. A., Pane, J. F., Ko, A. (2004) Natural programming languages and environments, *Communications of the ACM*, 47(9), p. 47–52.
- Navarro-Prieto, R., Cañas, J. J. (2001) Are visual programming languages better? The role of imagery in program comprehension, *International Journal of Human-Computer Studies* 54, p. 799 –829.
- Oliveira, O. L., Monteiro, A. M., Roman, N. T. (2011) From concrete to abstract?: problem domain in the learning of introductory programming, In: ITiCSE'2011: Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education, ACM Press, New York, NY, USA, p. 173–177.
- Pane, J. F., Ratanamahatana, C. A., Myers, B. A. (2001) Studying the language and structure in non-programmers' solutions to programming problems, *International Journal of Human-Computer Studies* 54, p. 237–264.
- Pattis, R., E. Karel the robot: a gentle introduction to the art of programming, 2nd ed., John Wiley & Sons, New York, 1995.
- Piaget, J., Inhelder, B. The psychology of the child, Basic Books, New York, USA, 1972.
- PPIG (2014) Psychology of Programming Interest Group, <http://www.ppig.org>, December 2014.
- Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E. Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B., Kafai, Y. (2009) Scratch: programming for all, *Communications of the ACM* 52(11), p. 60–67.
- SIGCHI (2014) ACM Special Interest Group on Computer-Human Interaction, <http://www.sigchi.org>, December 2014.
- SIGCSE (2014). ACM Special Interest Group on Computer Science Education, <http://www.sigcse.org>, December 2014.
- SIGITE (2014) ACM Special Interest Group for Information Technology Education, <http://test.sigite.hosting.acm.org/>, December 2014.
- SIGPLAN, 2013. ACM Special Interest Group on Programming Languages home page, <http://www.sigplan.org> , December 2014.
- Simon, B., Chen, T., Xavier, G. L., McCartney, R., Sanders, K. (2006) Commonsense computing: what students know before we teach (episode 1: sorting), In: ICER'2006: Proceedings of the 2006 International Workshop on Computing Education Research, ACM Press, New York, NY, USA, p. 29–40.
- Stefik, A., Hundhausen, C., Patterson, R. (2011) An empirical investigation into the design of auditory cues to enhance computer program comprehension, *International Journal of Human-Computer Studies* 69, p. 820–838.
- Summet, J., Kumar, D., O'Hara, K., Walker, D., Ni, L., Blank, D., Balch, T. (2009) Personalizing CS1 with robots, *ACM SIGCSE Bulletin* 41(1), p. 433–437.
- Villalobos, J. A., Calderon, N. A., Jiménez, C. H. (2009) Developing programming skills by using interactive learning objects, *ACM SIGCSE Bulletin* 41(3), p. 151–155.

Wicentowski, R., Newhall, T. (2005) Using image processing projects to teach CS1 topics, In: SIGCSE' 2005: Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education, ACM Press, New York, NY, USA, p. 287–191.

Xinogalos, S., Satratzemi, M., Dagdilelis, V. (2006) An introduction to object-oriented programming with a didactic microworld: objectKarel, Computers & Education 47(2), p. 148–171.