



HAL
open science

A Teaching System To Learn Programming: the Programmer's Learning Machine

Martin Quinson, Gérald Oster

► **To cite this version:**

Martin Quinson, Gérald Oster. A Teaching System To Learn Programming: the Programmer's Learning Machine. ACM Conference on Innovation and Technology in Computer Science Education 2015, Jul 2015, Vilnius, Lithuania. 10.1145/2729094.2742626 . hal-01238377

HAL Id: hal-01238377

<https://inria.hal.science/hal-01238377>

Submitted on 8 Jan 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Teaching System To Learn Programming: the Programmer’s Learning Machine

Martin Quinson
Université de Lorraine, F-54506, France
Inria, F-54600, France
CNRS, LORIA, UMR 7503, F-54506, France
martin.quinson@loria.fr

Gérald Oster
Université de Lorraine, F-54506, France
Inria, F-54600, France
CNRS, LORIA, UMR 7503, F-54506, France
gerald.oster@loria.fr

ABSTRACT

The Programmer’s Learning Machine (PLM) is an interactive exerciser for learning programming and algorithms. Using an integrated and graphical environment that provides a short feedback loop, it allows students to learn in a (semi)-autonomous way. This generic platform also enables teachers to create specific programming microworlds that match their teaching goals. This paper discusses our design goals and motivations, introduces the existing material and the proposed microworlds, and details the typical use cases from the student and teacher point of views.

1. INTRODUCTION

The Programmer’s Learning Machine (PLM) is a free integrated educational software environment for learning and teaching programming. The main target audience is college and university students that learn programming in loosely tutored practical sessions, but it can be used by younger pupils. It comes with 200 exercises covering the basics of programming, sorting algorithms and recursion.

Figure 1a shows PLM’s main window when discovering a new exercise. It is composed of two main panels: The left one presents the text of the mission, which provides any information needed by students to solve the exercise (ranging from the theoretical background and generic body of knowledge to practical details and hints) while the right panel displays a view of the current state of the exercise’s world. Each world in PLM entails at least one active entity that executes the student code. A student can interact with the entity using the interactive controls which are located below the world view to perform some preliminary experiments in solving the exercise.

An effective way to understand the goals of an exercise is to switch to the second tab of the world view as depicted in Figure 1b. It displays the state of the world as it should be at the end of the exercise. Pressing the “demo” button starts an animation of the operations leading from the initial state

to the expected one. The speed of this animation can be adjusted at will.

To pass an exercise, a student must type in the proper source code in the second tab of the left panel, as illustrated in Figure 1c. Any potential compilation or execution error is displayed in the console located at the bottom of the window. Then, the source code can be executed either continuously or step by step. At the end of the execution, if the world state does not match the final objective, an informative message explains the observed problem.

Exercises can be solved either in Java, Python or Scala, under the standard settings. The PLM uses the standard Java virtual machine in conjunction with either the standard Java or Scala compiler, or the standard Jython implementation of Python.

This short and graphical feedback loop improves the students’ motivation and makes their learning more effective in practice. Moreover, the PLM makes it easy for teachers to adapt the provided teaching material to their teaching goals, or to create new original material.

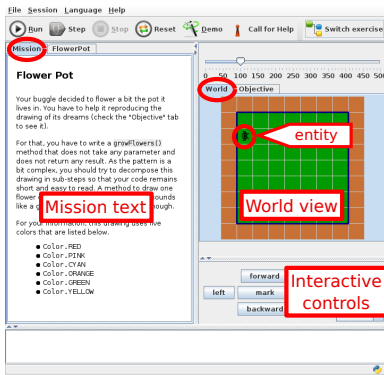
This paper is organized as follows: Section 2 discusses the design goals and rationale of the project. Section 3 situates the PLM within the state of the art. Section 4 shows the effectiveness of our educational tool through the presentation of the teaching material developed on top of it. Section 5 presents how teachers and resources authors can adapt and create additional resources. Section 6 discusses some evidences of the tool usage. Finally, Section 7 concludes the paper and points out some future work.

2. DESIGN GOALS AND RATIONALE

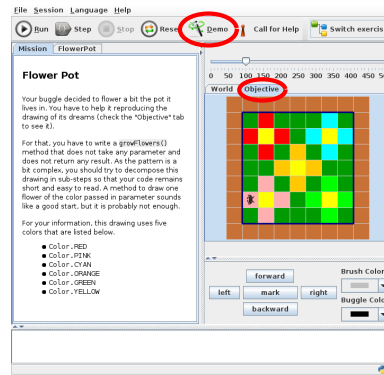
The overall rationale of the PLM is that programming is a learn-by-doing activity. Given the major role of practice in the learning of programming, the PLM intends to be an automated exerciser allowing students to practice with the programming tasks. The PLM is geared toward three categories of users, each of them mandating specific goals.

2.1 Toward students

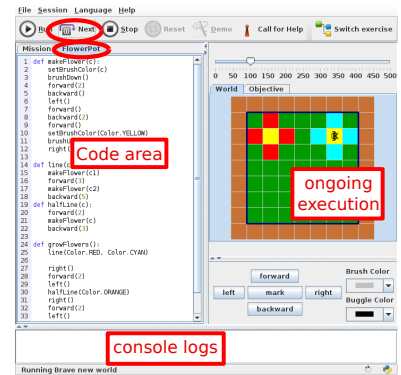
The PLM should facilitate autonomous learning and make programming engaging and satisfying. This main objective can be decomposed and detailed in several goals: Autonomous learning mandates *a consistent set of resources* with which the student can interact and experiment. Open-ended exercises enable free experimentation, but more guided exercises are precious to beginners that get easily paralyzed when facing choices. *A short feedback loop* provides a clear goal to the student’s experimentation while an



(a) Discovering a new exercise.



(b) Understanding the goal of an exercise through the use of the demo mode.



(c) Writing and executing code.

Figure 1: Basic use cases of the PLM.

automated evaluation leading to quick success experiences reinforces the student’s motivation. A *visual feedback* allows to focus on algorithmic concepts through the program effects, reducing the importance of syntactic elements. Finally, the tool should be *intuitive, easy to use and available* on most systems to avoid that technical details hinder user experience. Along the same lines, the environment must be translated and adapted to be usable by non-native English learners. This is particularly true for younger pupils.

Another important design goal of the PLM is to ease the transition to other programming contexts through the use of standard tools and languages. *Several programming languages are proposed* in the environment, so that the user is free to choose. Generic purpose programming languages are however known to be less adapted to novice programmers [2]. Java programs must for example use `public static void main(String[] args)` as an entry point, exposing beginners to advanced notions on day one [13]. Even if this code can be provided to students, it induces an extra cognitive burden that can drown beginners. It is then preferable to *template the provided code and hide* the parts that are less relevant to the teachings.

2.2 Toward authors of resources

The PLM should provide technical support to ease the creation of innovative learning situations adapted to each context. In particular, it should be easy to build a new form of microworld providing a specific learning situation. Moreover, the environment should provide all non-functional code, for instance related to user interactions, code compilation or user sessions persistence. An integrated instrumentation infrastructure would allow to assess the efficiency of the developed pedagogical resources.

2.3 Toward teachers

The environment should provide convenient and ready to use tools without hindering the practitioner’s pedagogical freedom. It is expected that most teachers will leverage existing resources. This is similar to the classical use of existing books that is common in most teachings. Nevertheless, it should be easy for teachers to adapt the existing resources and assemble them to build their own learning path on top of the existing resources.

3. CONTEXT AND STATE OF THE ART

The PLM builds upon the concept of Programming Microworld, introduced in the eighties through the LOGO programming language [10]. In such settings, the student controls an *entity* or actor that interacts with its *environment*. In the initial LOGO microworld, the entity is a robotic turtle that leaves trails on a sheet of paper as it walks. Many systems build upon this idea, such as LogoBlocks [1] or the LEGO Mindstorms System.

One drawback of the LOGO microworld is the scarce interactions between the entity and its environment: there is no way for the turtle to check the state of the sheet. Subsequent microworlds provide richer interactions. The most popular is certainly *Karel the robot* [11], where the actor, a robot named Karel, evolves in a world consisting of intersecting streets and avenues. It can pick and drop “beepers” (objects laying on the ground) but cannot pass the walls. This leads to richer interactions through predicates that can test the presence of walls and beepers. Kara [7] is another microworld where a ladybird picks leaves while avoiding trees. Many other similar microworlds have been proposed in the literature, as reviewed in [2].

In addition to such generic microworlds, specialized microworlds can introduce specific concepts. In [15], the authors leverage three microworlds: Most of the teaching are conducted with the *BuggleWorld*, a generic microworld with rich entity-environment interactions. Recursion is first introduced using the *PictureWorld* that allows the construction of complex quilt patterns by combining simple shapes with basic transformations. Recursion is then further exercised through the drawing of polygons, spirals and trees in a LOGO-inspired microworld. This idea is pushed further in [3] where each exercise leverages a specific microworld. The limit of this approach is that the microworlds of both projects are developed separately without any code reuse. The non-functional code then induces an important extra burden. By contrast, the PLM factorizes this code to ease the authoring of pedagogical resource.

Many other similar environments have been proposed in the literature. In the taxonomy of Kelleher and Pausch [8], the PLM is within the category *Teaching System/Mechanics of Programming*. It covers several sub-categories: Simplify Typing Code; Making New Models Accessible; Tracking Program Execution and Make Programming Concrete.

Scratch, Alice and Greenfoot are very renowned educational projects to teach and learn programming. Scratch [12] enables kids to develop interactive stories and animations. Since scripts are constituted of building blocks that are visually assembled, it is accessible to kids even before they master reading and writing. The Alice [5] environment teaches older pupils about object-oriented programming through the design of scripted 3D animations. Greenfoot [9] uses the standard Java language for a similar goal.

The main difference between the PLM and these projects consists in the ability to propose auto-evaluated resources. Scratch, Alice and Greenfoot can be considered as IDEs that are specifically tailored to learners. However, in the PLM, teachers can prepare specific exercises on which students work autonomously, at their own pace. Both approaches are complementary: learners could first start with guided resources on the PLM to learn the concepts they need, and then move to an educational IDE to build their own projects using these concepts.

4. EXISTING TEACHING MATERIAL

This section presents the teaching resources developed on top of the PLM and distributed with the environment itself. The main goal of this discussion is to demonstrate the effectiveness of the PLM as an educational environment. The presented resources are interesting per se, even if we have no formal evidence of their intrinsic effectiveness. Many of them were previously existing in the literature. The practical effectiveness of the PLM (and to some extent of these resources) is discussed in Section 6.

4.1 The PLM’s Universes

Since the PLM entails several kind of microworlds, we refer to each family of microworlds as a *universe*. We now detail the existing universes, focusing on their pedagogical features and on their typical use in exercises.

The Buggles.

Originally, this generic universe was the only existing universe in the PLM. It relies on an original idea of Franklyn Turbak at Wellesley College [15]. It features *the buggles*, that are able of many interactions with their environment: they can pick and drop objects, paint the ground, read and write messages on the ground, hit walls, etc.

Thanks to its versatility, this universe is used in many proposed exercises.

Sorting Algorithms.

Several original universes are specifically tailored to teach sorting algorithms. The first one targets the exploration of sorting algorithms. The demo mode can be used at full speed to get a practical idea of the different complexity of algorithms, or to understand the specific behavior of an algorithm step by step.

Interestingly, it is not sufficient to sort the provided array of data to pass a typical sorting exercise, but the proposed solution must access the data exactly the same amount of time than the expected solution. This is enforced through adapted primitives that mediate and count any data access. Using `isSmaller(i,j)` to compare cells does account for two reads. Using `copy(i,j)` or `swap(i,j)` to modify the data accounts respectively for one read plus one write, or for

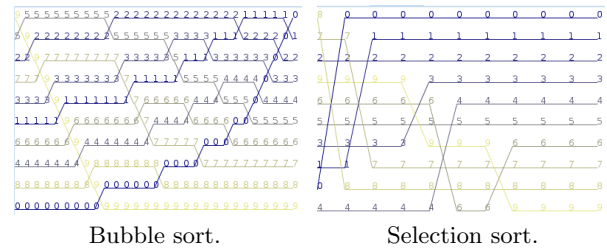


Figure 2: Temporal view of the sorting world.

two reads plus two writes. This constraint on the amount of data accesses forces students to strictly implement the expected algorithm, which in turn requires a very good level of understanding of the presented algorithms.

When the amount of data accesses does not match, understanding the difference between the proposed code and the expected solution can be difficult. To ease this process, student can graphically explore the history of their sorting algorithm as shown in Figure 2. The time is represented as abscissa while ordinate represent the array cells. The lines represent the values in these cells; When two lines cross, this means that two values were swapped at this timestamp. This representation, introduced by A. Cortesi¹, is precious to understand the behavior of sorting algorithms.

The *Rainbow Baseball* is another PLM universe that applies the classical sorting algorithms in a funny and original setting. It builds upon the Pebble Motion Problem [4], where objects must be placed on the right vertex of a graph under several movement constraints. This problem is similar to the 15-puzzle problem. This engaging universe can be played by clicking on the pebbles to move them, but the provided lesson requires students to reimplement the major sorting algorithms. A temporal view allows to compare the proposed code with the expected one.

The *Pancakes Universe* builds upon the classical pancake problem [6], where a stack of pancake should be sorted only by flipping pancakes at the top of the stack. The corresponding lesson entails several existing algorithms to solve this problem, some of them being non-trivial.

The *Dutch Flag* universe is a simple universe whose solution is a linear sorting algorithm adapted to sort an array containing only three kinds of values.

Unit Testing.

This universe is very specific as it does not provide any graphical representation. Each exercise proposes a method prototype that the students must fill in. This method is then tested against a comprehensive test of parameter values. This universe, inspired from the work of Nick Parlante², is probably less motivating than graphical microworlds, but it is efficient to practice on a particular point. The introductory lesson provides for example short exercises on conditionals where the students must write simple boolean expressions matching the provided textual descriptions.

The LOGO Turtles.

This universe reimplements the classical LOGO microworld. It is used in two of the proposed lessons. In the

¹<http://corte.si/posts/code/visualisingsorting/>

²<http://codingbat.com/>

recursive lesson, the students are asked to draw recursively polygons, spirals, trees and fractals. The “turtle art” lesson presents several classical LOGO drawings to inspire the students, that are expected to then play in creative mode where their solution is not checked against any objective.

Recursive Algorithms.

Recursive Lists are a classical data structure, even if vectors are more used in Java and Python. In Scala, the classical List datatype is used while a dedicated type is proposed in Python and Java. 17 classical exercises are proposed in this universe, such as computing the length or finding one element.

The *Hanoi Towers* problem is classically used to introduce recursive programming. In the PLM, 12 progressive exercises explore some less known variations of this problem to exercise the recursive decomposition abilities of the learners.

4.2 Proposed Learning Path

The exercises are not isolated in the PLM, but grouped by thematic lessons providing a coherent progression on each topic. This is particularly important to self-learners using the unmodified tool, even if most teachers will probably adapt this path to their settings.

Overall, more than 200 exercises are proposed in the PLM. Their corrections constitute a body of roughly 3000 lines of code (per programming language). The fastest students need around one week full-time to solve all the exercises.

4.2.1 Tactical Programming

The first lesson introduces the basic concepts of programming to absolute beginners. After a quick tour of the environment, the notion of instructions is presented through very simple Buggle exercises. The lesson then introduces conditionals and while loops. The concepts are presented as part of simple exercises and reinvested immediately in several application exercises that leverage the universes’ versatility. An extra complexity comes from the fact that the pattern matching of Scala is more powerful than the switch-case construct of Java, while Python offers no switch construct. The mission text are then dynamically adapted to the chosen programming language so that the right concept and syntax are presented to the student.

Variables are the next introduced concept, before for loops and do-while loops. In python, we present the idiomatic way toward do-while loops since this construct does not exist in this language. 20 exercises are proposed for these notions, 15 of them being engaging and situated application exercises that follow uncluttered introduction exercises presenting the concepts. Some exercises induce more complex loop settings, with non-trivial loop conditions, limit cases and modeling task to identify the repeated instructions, as advised in [14].

Functions are then introduced, followed by methods and parameters. Functional decomposition is introduced through several exercises requesting to repeat a given pattern an increasing amount of time. About 15 exercises introduce and apply these concepts. Most of our students need 4 to 6 hours to reach the end of this unit.

About 40 unit testing exercises to practice with writing of simple conditionals take place at this point of the progression. They could theoretically be placed earlier, but our unit universe requires the student to know about meth-

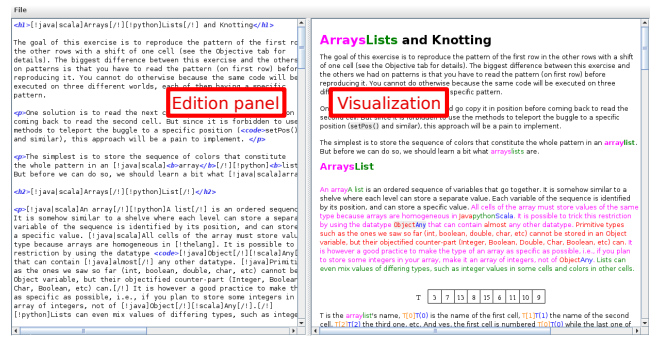


Figure 3: Mission text editor.

ods. Sequence of elements (arrays in Java and Scala, lists in Python) are then explored through 18 exercises.

This concludes the introductory lesson, that is usually completed by absolute beginners at university level in 10 to 20 hours. It can be followed by two application lessons: the first one quickly explores classical maze algorithms while the other introduces a simple form of cellular automaton called Langton’s ant. This is easy to implement for the students, and leads to interesting drawings.

4.2.2 Algorithmic Basics

After the language syntax and basic constructs introduction, we move on to more algorithmic lessons. One lesson explores the classical sorting algorithm, while two other lessons situate these algorithms in the engaging microworlds presented in previous section. Another lesson explores recursion through recursive lists, Logo-based figures and variations of the Hanoi Towers problem.

5. EXTENDING THE PLM

Teachers can reuse the proposed material as-is, or they can adapt and extend it. One current limitation of the PLM is that such adaptation can only be done in Java for now, forcing teachers to have some notions of this language even if they teach Scala or Python.

5.1 Creating new lessons

The easiest PLM adaptation is to create a lesson that builds a new learning path from existing resources. This is trivially done by building a Java collection of the selected resources during the lesson’s initialization. This could be further simplified in the future by using configuration files instead of Java code.

5.2 Creating new exercises

A typical exercise is composed of a mission text, the description of the initial environment’s state, and a correction entity implementing the requested work.

The mission text is an HTML file, possibly with conditional inclusions to adapt the text to the selected programming language. Since such conditional texts tend to be intricate and burdensome to edit, the PLM provides an adapted text editor (Figure 3). The text is edited in the left panel and previewed in the right panel. A color code depicts graphically the parts that are specific to each programming language. The mission text can also entail optional *hints*, that are shown only on explicit student request.

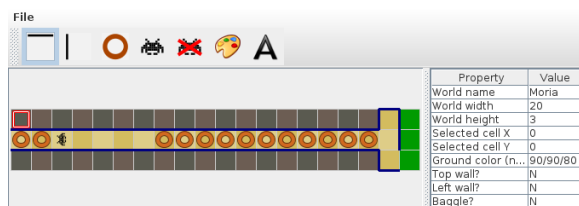


Figure 4: Buggle world map editor.

One way to build the initial state of the microworld is through a dedicated Java class whose complexity naturally depends on the world to instantiate. A graphical editor is provided for buggle worlds, that can become rather complex. This intuitive tool is depicted in Figure 4.

Most exercises test students' code on multiple instances of the problem to detect more errors. From the author perspective, this simply requires to attach several world instances to the exercise.

The correction entities have several roles: They are applied to the initial worlds to compute the objective worlds; They are executed when playing the demo, and they provide an hosting scaffold around the student's code chunk to produce valid files. This last usage requires to automatically edit the entity. To this end, the entity source code must be annotated as shown in Figure 5. When compiling the student's code, the section marked with `BEGIN/{end template}` is removed and replaced with the student code while the other parts are left unchanged. Once the `BEGIN/{end solution}` section is removed, the `BEGIN/END TEMPLATE` section is also used as initial content in the source code editor. Here, this mechanism pre seeds the editor with the function's prototype that must be filled in by the student.

Since they are used as a scaffold around the student's code, it is necessary to write one correction entity in each programming language for each exercise.

5.3 Designing a New Microworld

Adding a radically new learning context to the PLM requires to extend the classes defining an universe: the *World* class contains the microworld state and data; the *Entity* class is the ancestor of all correction entities. This class is mainly responsible to adapt the World interface and provide to the user convenient primitives to alter the world. The

```
public class SlugTrackingEntity extends SimpleBuggle {
    // Some additional code adapted to this exercise
    /* BEGIN TEMPLATE */
    boolean isFacingTrail() {
        // Write your code here
        /* BEGIN SOLUTION */
        if (isFacingWall())
            return false;
        forward();
        boolean res = getGroundColor().equals(Color.green);
        backward();
        return res;
        /* END SOLUTION */
    }
    /* END TEMPLATE */
}
```

Figure 5: Example of annotated entity source code.

WorldView provides a graphical representation of a given World. Any universe should also be documented through an HTML following the same conventions than the mission texts of exercises. Finally, most universes should entail a *WorldPanel* class that provides the entities' interactive controls.

Universe	#lines	Universe	#lines
Buggle	1400	Panecake	500
Baseball	1000	Units	500
Sorting	900	Hanoi	500
Turtle	700	Dutch Flag	400

Table 1: Complexity of the main universes.

Table 1 presents the code complexity of each universe. The numbers remain surprisingly low despite the wealth of the proposed universes, proving the effectiveness of the code factorization enabled by the PLM.

6. TOOL ADOPTION

We use the PLM since seven years in our institution. Thanks to it, absolute beginners become quickly confident in the programming syntax, allowing to take on more advanced content in the same time than before its introduction. Although only spread by word of mouth so far, the environment is proposed as a complement to the regular teaching in a few institutions.

Figure 6 presents the feedback collected from our students after the semester. The evaluation was strictly anonymous; we received 53 responses from 100 enrolled students. Our Lickert scales did not propose any neutral choices to get more critical feedback. In response, some students did not provide any answer on controversial questions forcing us to consider the absence of answer as a neutral choice.

The perception of the educational value of the PLM is overwhelmingly positive, with only 3 students disagreeing and 34 students strongly agreeing with this statement. The entertaining value was also very appreciated although less strongly, with 31 students agreeing on this statement and only 16 strongly agreeing. The motivational aspect is also very well received as over 80\students agree or strongly agree that practical sessions are more motivating when using the PLM than with classical sessions. The actual educational value is more discussed as one third of the students report that they did not progress while using the PLM. This may be linked to the fact that some of the students that are requested to follow this teaching can already program beforehand. As noted previously, even these students find the PLM useful and entertaining to some extent.

After these questions on specific qualities, we evaluated the overall feeling of the students. Consistently with previous results, the vast majority would advice the environment to a friend. The last question was meant as a test of the students' objectivity. We asked whether they thought of the PLM as a good argument for potential candidates to our school. This statement is very arguable as the PLM is used only in very few teachings of the curriculum (even if students enrolled in the first year may oversee this fact). One third of the students disagreed with this statement, and one fourth skipped this question. This statement was thus very mildly agreed, reinforcing our confidence in the objectivity of this evaluation.

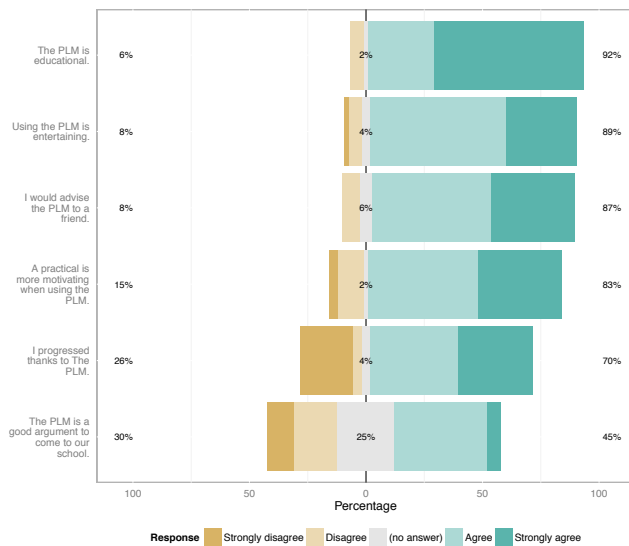


Figure 6: Anonymous Evaluation of the PLM by our Students (54 answers).

7. CONCLUSION AND FUTURE WORK

In this paper, we described the Programmer’s Learning Machine, an integrated educational software dedicated to the learning and teaching of programming. It is designed for users with no or little programming experience and aims at providing practice in a friendly and attractive environment. An extensive but coherent set of exercises is presented to students, who can progress at their own pace. Several microworlds are leveraged, some of them specialized in specific programming concepts. The exercises can be solved using either the Java, Scala or Python programming language, using the standard compilers. The tool and all teaching materials are fully translated to English and French (and partially to Italian and Portuguese). The proposed lessons entail over 200 exercises, covering the imperative kernel (basic syntax and control structures), sorting algorithms and recursion.

The PLM is freely available from the project’s web page³, under an open-source license. It is also integrated to the Debian distribution of Linux. It is known to work on Windows, Linux and MacOS. We are now actively fostering the emergence of a user community.

Future work naturally entails the extension of the proposed teaching material and the addition of new microworlds to explore other concepts such as backtracking, dynamic programming, or classical data structures (heap, stacks, graphs). We are currently introducing an instrumentation infrastructure that could be used to quickly detect the students requiring the teacher’s help. Such infrastructure could also be used to study the effectiveness of specific teaching strategies and approaches. This will enable formal studies of the apparatus efficiency that will in turn help improving further this educational environment.

8. REFERENCES

- [1] Andrew Begel. Logoblocks: A graphical programming language for interacting with the world. Master’s thesis, Massachusetts Institute of Technology, 1996.
- [2] Peter Brusilovsky, Eduardo Calabrese, Jozef Hvorecky, and Philip Miller. Mini-languages: A way to learn programming principles. *Education and Information Technologies*, 2(1), 1997.
- [3] Nadya Calderon, Jorge Villalobos, and Camilo Jimenez. Developing programming skills by using interactive learning objects. In *14th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE’09)*, Paris, France., July 2009. ACM.
- [4] Gruia Calinescu, Adrian Dumitrescu, and Janos Pach. Reconfigurations in graphs and grids. In *LATIN 2006: Theoretical Informatics*, volume 3887 of *Lecture Notes in Computer Science*. Springer, 2006.
- [5] Matthew Conway, Steve Audia, Tommy Burnette, Dennis Cosgrove, and Kevin Christiansen. Alice: lessons learned from building a 3d system for novices. In *CHI ’00: Proceedings of the SIGCHI conference on Human factors in computing systems*, New York, NY, USA, 2000. ACM.
- [6] William H. Gates and Christos H. Papadimitriou. Bounds for sorting by prefix reversal. *Discrete Mathematics*, 27(1), 1979.
- [7] Werner Hartmann, Jurg Nievergelt, and Raimond Reichert. Kara, finite state machines, and the case for programming as part of general education. In *Proceedings of IEEE Symposium on Human-Centric Computing Languages and Environments*, Stresa, Italy, 2001. IEEE Computer Society.
- [8] Caitlin Kelleher and Randy Pausch. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Comp. Survey*, 37(2), 2005.
- [9] Michael Killing. The Greenfoot Programming Environment. *ACM Transactions on Computing Education (TOCE)*, 10(4), November 2010.
- [10] Seymour Papert. *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books, 1980.
- [11] Richard Pattis. *Karel the Robot: A Gentle Introduction to the Art of Programming with Pascal*. John Wiley and Sons, London, 1981.
- [12] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. Scratch: Programming for all. *CACM*, 52(11), 2009.
- [13] Eric Roberts, Kim Bruce, Kim Cutler, James Cross, Scott Grissom, Karl Klee, Susan Rodger, Fran Trees, Ian Utting, and Frank Yellin. The acm java task force project rationale. Technical report, ACM, 2006.
- [14] Mara Saeli. *Teaching Programming for Secondary School: a Pedagogical Content Knowledge Based Approach*. PhD thesis, Technische Universiteit Eindhoven, 2012.
- [15] Franklyn Turbak, Constance Royden, Jennifer Stephan, and Jean Herbst. Teaching recursion before iteration in CS1. *Journal of Computing in Small Colleges*, 14(4), may 1999.

³<http://www.loria.fr/~quinson/Teaching/PLM/>