GrandeOmega
A didactic model for learning programming

Francesco Di Giacomo {francesco.digiacomo@unive.it}[1]
Mohamed Abbadi {mohamed.abbadi@unive.it}[1]
Giuseppe Maggiore {giuseppe@grandeomega.com}[2]
Marijn Bom {marijn@hoppinger.com}[2]

[1]Universita' Ca' Foscari of Venice
[2]Hoppinger

Abstract

Teaching (and learning) how to program is an open problem. Both teachers and students cite a variety of issues, ranging from difficulty bridging the gap between theory and practice, to lack of simple assignments suitable for beginners. We believe that technology can offer a powerful tool in supporting a solution to these problems, but at a high cost in internal complexity. For this reason we have built GrandeOmega: a generic, web-based code-interpreter that can be easily used to build interactive lectures and assignments where students experiment with learning how to program while receiving real-time feedback. The experimental results show that GrandeOmega generally boosts the overall performance of students, increasing in some cases also the pass rate of classes.

1 Introduction

Learning how to program is a daunting task. There are several studies [4, 5, 7, 11, 12, 13] documenting all sorts of didactic approaches in teaching programming in both lower and higher education by means of games, flipping the classroom, role playing, massive online courses, and much more.

Unfortunately, a clear winner has not yet emerged. The broad variety of methods is perhaps a symptom of a still open problem, which we believe is still very much present and made even more urgent by government calls to increase the number of graduates [2] to keep driving knowledge economies.

We observe that in many institutions, either the survival rates of students are quite low [1], or (in some extreme cases) the quality of the curriculum is adjusted downwards to avoid punishing students too much. Feedback from companies around the authors is often quite damning: the quality of graduate programmers is often too low, citing lack of understanding of basic aspects of

programming such as memory models, concrete semantics of languages, types and type systems, with the result of slow, bug-ridden code being shipped all too often [10, 9]. In some places we are even witnessing a slow fading of the engineering from software engineering.

We propose that learning to program is indeed fundamental, but it must happen at a high level. Those who will build the digital infrastructure of tomorrow must be able to do so with reliable, high-performance results: an infrastructure that barely works and holds together will be worthless.

Problem statement: the broader issue that we try to tackle in this paper is the study and improvement of teaching programming in higher education institutions.

To do so, we will begin by analysing the issue of how complex is it to learn how to program (Section 2). We will then provide a sketch of our solution with all the ingredients we believe to be fundamental for solving the issue (Section 3). We take the ingredients together and discuss the architecture of our actual implementation of such a solution (Section 4). Finally, we provide an evaluation of the impact of our (large) trial at the Rotterdam University of Applied Sciences (Section 5).

2 Why is it so hard?

Within the context of higher education, learning programming is hard for both beginners and students with past experience.

It is suggested by some [15] that learning programming is no different than most other complex skills: it takes roughly ten years (ten thousand hours) to become truly proficient.

The reason why it takes so long is disarmingly simple. Programming requires both the ability to *understand* and to *design* code.

2.1 Understanding code

Understanding code is a passive skill, but not any simpler because of it. The true meaning of code is the sequence of steps that the machine will actually perform: every single bit that will be read and written as a result of an instruction is part of the meaning of that instruction, just like every cache hit-or-miss, the activation of the CPU ALU(s), network channels, operating systems, interpreters, just-in-time compilers, and ultimately interactions with users. Being able to figure precisely what a program does, and how it does it (also in terms of performance) requires the ability to formulate an abstract idea of the program behaviour, and the mapping of this *abstract idea* to the concrete components when more specific reasoning is needed.

The sheer size of the machinery involved in the execution of even the simplest program is simply *very large*, and *the ability to think hierarchically and zoom in and out of the details as needed takes a lot of experience.*

2.2 Designing code

Designing code is an active skill, and as such intrinsically complex. Designing code requires the formulation (and therefore the choice) of a design strategy, usually in a top-down fashion, which is then recursively turned into a more and more concrete definition of the program. Being able to design a program effectively requires the ability to choose a specific design among a series of possible designs, which taken together form *the abstract meta-strategies* that characterise a programmers knowledge, style, and experience.

The sheer size of the design space of even the simples program is *so large as to be essentially infinite* (we are talking about finite machines after all!), and the ability to *formulate meta-strategies and employ them recursively takes a lot of experience.*

2.3 Size matters (and so does structure)

We believe the size of the domain to be the core of the issue. Even though some students might already know a few tricks to produce working programs in some very narrow domain, the fundamental ability to abstractly reason about code (both for understanding and designing programs) is usually severely lacking in first year students.

Moreover, we cannot just solve the problem by throwing unstructured assignments such as read this code or try and write this program, as we must train the specific mental activities that we wish to stimulate in students. Specific training must be structured in order to gently guide the activation of the proper thought structures, in a slow buildup of complexity and freedom to express ones own creativity.

2.4 The issues

We close this session by identifying a series of practical, concrete issues that we believe sum up the discussion so far:

ID	Issue
REASON_MODEL	Students need extensive practice with reasoning about models of programs
REASON_DESIGN	Students need extensive practice with reasoning about existing design strategies
EXTEND_DESIGN	Students need extensive practice with extending existing programs (which should follow a formative design).
EXTEND_BUILDUP	Students need a buildup in complexity with their extension activities.

Table 1: Issues about learning programming

By keeping these issues in mind, we will now setup a proposed didactic model which we believe can mitigate them or outright resolve them.

3 The ingredients and the didactic model

The didactic model we propose (just like the implementing software) is called GrandeOmega.

Whereas many revolutionary didactic models completely remove emphasis from the role of the teacher, thereby forsaking the richest source of knowledge and explanation available in the vicinity of students, GrandeOmega sets out to empower both teacher and student.

The teacher sets up his course, in a way similar to traditional lectures. Each lecture is made up of slides, and these slides follow some visual formalism to explain code. One of the most used such formalism is (recursive) tables of names and values. Traditional slides are *passive*.

Some of the slides, ideally one every quarter to half an hour, are active slides. The active slides use the *very same formalism of the passive slides, but with a major difference: students can experiment with them.* This means that the theory discussed by (and with) the teacher is not separated from the practice, but comes to life and strengthens the practice just as much as the practice gives context and use to the theory.

Experimentation within active slides takes two forms: *forward* and *backward* assignments.

Forward assignments Within forward assignments students practice their understanding of code. They get to see whole (small) programs, and they are asked by the system to provide the values of variables or constants (`REASON_MODEL` and `REASON_DESIGN` issues from Table 1). The values that need to be predicted by students can range from the integer value of a global variable (`x=1`), to the value of a reference in the virtual heap (`l=ref(10)`).

Backward assignments Within backward assignments students practice with design strategies to build code. They get to see whole (small) programs where some bits have been hidden, and they also get to see all the steps that the complete program would take. Students must guess back the hidden bits of code (`EXTENDED_DESIGN` and `EXTENDED_BUILDUP` issues from Table 1). The hidden bits of code range from a constant (`5`) or a variable name (`x`) to whole expressions (`i*2`), to whole instructions, functions, methods, classes, queries, potentially up to the whole program.

Immediate feedback Each and every mistake automatically produces immediate, visual feedback in the form of colors and icons. The input from students *is evaluated, not just compared as text.*

Gamification Progress of a student, and every success, is stored and clearly shown in context by means of colors and icons. A student can see, at a glance, how far he is: whether lagging behind, or following nicely.

Insight to teachers All the *data gathered* by the logging systems is *analysed and presented to teachers*. This makes it possible to identify difficult topics (assignments where the majority of students is getting stuck), identify struggling students (who are getting stuck in most of the assignments), and to identify negative behaviours (too few hours of study, lack of regularity, etc.). As stated above, we strongly believe that modern didactics should not try to provide a revolutionary removal of the teacher: *the teacher remains the director of the educational orchestra, even though lots of active work is performed by the students*.

Extra ingredients Understanding and designing code on a smaller scale, with a gradual buildup in size and complexity, makes it possible for students to gain a deep understanding of the underlying logical mechanisms.

To further increase efficiency of a curriculum, the task of learning should be cornered from multiple sides. Students should also be presented with the challenge of freely experimenting with open designs and projects to build. Thus, the didactic method as a whole can be summarised as:

- forward assignments to learn *understanding*;
- backward assignments to learn *design*;
- projects to *sum it up*.

4 Technical details

The web-based implementation of GrandeOmega [6] features, at its core, a generic code interpreter which takes as input a specification of one or more programming languages, and yields the following as output:

- an interactive, assistive code-editor (Figure 1);
- a visual debugger (Figure 1);
- a forward assignment editor (for teachers only - Figure 2);
- a backward assignment editor (for teachers only - Figure 2);
- a slide editor (for teachers only - Figure 3);
- a slide presenter (for teachers and students);
- a forward assignment environment (for students only - Figure);
- a backward assignment environment (for students only).

The system also features a dashboard which shows assignment data over a whole course, a whole class, or a single student.

Figure 1: Code editor and debugger

Figure 2: Teacher's assignment editor

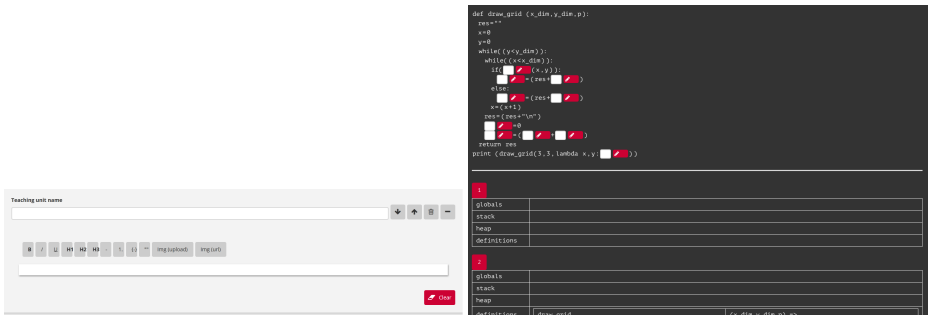

Figure 3: Slide editor and student's editor

4.1 Architecture

In order to ensure the right level of responsivity, the application is features a large front-end written in React and TypeScript. React is a UI framework published by Facebook, implementing the functional reactive programming paradigm [3, 8] in JavaScript and for browsers. The combination of referential transparency and controlled state updates make it possible to build truly complex, deeply hierarchical interactive structures with a high degree of correctness and confidence[add reference to referential transparency correctness]. TypeScript is an extension of JavaScript built by Microsoft, featuring a rich type-system inspired from the world of statically typed functional languages, emphasising inductive composition of types and generic programming. The com-

bination of (functional) reactive programming and a rich language of statically checked types has enabled us to build a complex system with high performance and availability, without having to leave the platform of the web and therefore without sacrificing accessibility.

The backend is twofold: on one hand we have a traditional Ruby on Rails application which is quite simple in its data storage and retrieval operations. On the other hand, the same code interpreter that powers the front-end is invoked by the backend in the form of a Nodejs service (Nodejs is a server-side JavaScript execution environment). By means of Nodejs, both the frontend and the backend feature the very same code interpreter, avoiding very unpleasant mismatches such as a correct answer on the front end which maps to a wrong answer according to the back end, and viceversa.

4.2 The generic code interpreter

The code interpreter is the beating heart of GrandeOmega. First of all, there is no single programming language supported: GrandeOmega is extensible to support any programming language definable: from Python (currently implemented), to SQL, and further to mathematical languages such as the lambda calculus or even languages for set theory and boolean logic.

The code interpreter is based on pattern matching and substitution and it directly mimic the operational semantics [14] used to describe the formal semantics of computer languages. The operational semantics define a set of rules in the form of logical rules, which means they are made of a (optional) set of premises that, if satisfied, will produce the result defined in the conclusion. If a rule does not contain premises it is called *axiom* (meaning that its evaluation is immediate). Below you find an example of such rules describing the evaluation of the sum of two arithmetic expressions:

$$\frac{}{\langle \texttt{Integer x} \rangle \Rightarrow \texttt{Integer x}}$$

$$\frac{\begin{array}{c}\langle \texttt{left} \rangle \Rightarrow \texttt{Integer x} \\ \langle \texttt{right} \rangle \Rightarrow \texttt{Integer y}\end{array}}{\langle \texttt{left} + \texttt{right} \rangle \Rightarrow \texttt{Integer (x + y)}}$$

The first rule is an axiom: if the input of the rule is an integer number `x` then its evaluation simply returns itself. The second rule evaluates the sum of two expressions by recursively calling the evaluation on the left and right expression. If the evaluation succeeds and returns an integer value, then the result of the whole rule is the arithmetic sum of the result of evaluating the two expressions.

From this example we proceed to define the general way of evaluating a semantics rule:

- The left part of the conclusion is analysed by using pattern matching in the following way:

  - If the current element is a variable then the pattern matching succeeds.
  - If the current element is a constant we compare it with the input of the rule and if they are the same then the pattern matching succeeds.
  - If the current element contains operators or other syntactical structures of the language (like the plus operator in the example), we compare the structure of the input of the rule with the one in the conclusion. This means that we check if the syntactical structure of the input of the rule is the same in the conclusion and then we check each argument of the syntactical structure by recursively applying these pattern matching steps.

- We try to evaluate each one of the premises in the following way:
  - We try to run each semantics rule in the language definition until one returns a result.
  - If the result of a premise (right part of the arrow) is a specific syntactical structure then we test the pattern matching.
  - If all the semantics rules fail to produce a suitable result then the current semantics rule fails to return a result.

- We generate the result of the current semantics rule.

4.3 The generic structured code editor

The code interpreter is not only used to implement the operational semantics of a computer language, but also to implement an interactive parser for the code editor. Let us consider the statement

```
if expr statements else statements
```

The interpreter contains the definition for the language construct, so it is able to detect the terminal and non-terminal elements of the statement. The code editor uses, in the current version, an infix notation, so for example the expression `5 + 3` written using the suffix notation must be typed with the prefix notation `+ 5 3`. When the user types a valid terminal symbol, the code editor will detect it and generate the remaining keywords as text and display them on the screen. The non terminal symbols (like `expr` and `statements` in the example above) will be generated as input boxes. The user will be able to further edit the code inside these boxes, which will be recursively evaluated with the same method. For instance, the following code

```
if x > 5
        x = x + 1
else
        x = x - 1
```

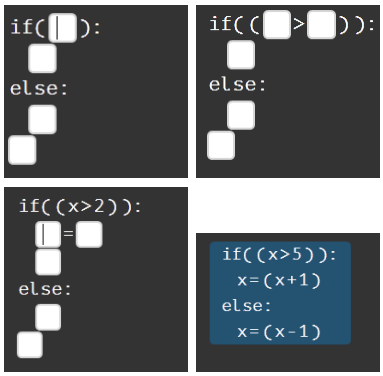

Figure 4: Code editor example: (1) `If` creation, (2) condition creation, (3) variable assignment creation, (4) final result

will be created by initially typing `if` in the code editor (followed by a space to mark that the keyword is over). In that moment the editor will lookup the keyword in the language definition and create the appropriate layout (Figure 4 (1)). The editor will display the keywords `if` and `else` as text, and create 3 input boxes for the condition, the `if` block, and the `else` block. In order to create the condition we must type the comparison by starting with the `>` symbol, which will create again two input boxes by detecting that a keyword has been typed and that the left and right arguments of that keyword are non-terminal symbols (Figure 4 (2)). Analogously, it is possible to create the variable assignment (`=`) and the assigned expressions (containing the keywords `+` and `-`). Figure 4 (3) and (4) summarizes these steps.

5 Evaluation

GrandeOmega has been tested extensively with students from Hogeschool Rotterdam, a university of applied science in the Netherlands. The classes were divided into two groups: in the first classes were given some programming assignments to be completed in the traditional way (without GrandeOmega), while in the second other classes were asked to solve the assignments both in the traditional way and in GrandeOmega. Table 2 and Figure 6 contain data relative to the pass rate and average grades of the classes that were asked to used also GrandeOmega, with and without using it. Table 3 and Figure 5 contain data relative to the accuracy of the prediction on the student success performed by GrandeOmega. The total number of students who participated is 241.

The data shows that the use of GrandeOmega enhanced the pass rate of classes 1B and 1C, but not that of 1F, 1L, and 1A. This means that the students of 1B and 1C failed questions in the traditional way that they were able to solve in GrandeOmega. Nonetheless, we can see that it always enhanced the grades of all the classes by at least 4%. Moreover, if we compare the average passing

grade of these classes with those who never used GrandeOmega, we can see that we reach a difference of even 12%.

GrandeOmega was revealed to be effective even in predicting the success of students with a total reliability of 77%, based on the percentage of assignment completed correctly.

Class	Completions	Pass rate	Pass rate G.O.	Average grade	Average grade G.O.
INF1B	71.8	3	14	84.7	97.2
INF1F	56.7	6	5	77.0	88.1
INF1L	48.1	2	2	75	83.8
INF1A	41.7	7	7	82.1	86.5
INF1C	35.6	5	7	82.5	93.3

Table 2: Student performance before and after GrandeOmega

Class	**Correct prediction**	**False positive**	**False negative**	**Incorrect prediction**
INF1B	16	8	2	10
INF1F	11	2	4	6
INF1L	18	3	1	4
INF1A	13	0	2	2
INF1C	15	1	1	2

Table 3: Prediction of student performance

Class	**Average grade**	**Passed students**	**Avarage passing grade**
INF1H	41.4	3	83.3
INF1E	30.6	1	75
INF1J	23.3	0	N.A.
INF1G	41.4	7	80.3

Table 4: Results of classes without GrandeOmega

6 Conclusion

In this work we addressed the problem of teaching and learning how to program and we proposed the use of a didactic model called GrandeOmega. The web implementation of GrandeOmega was tested on a subset of students from Hogeschool Rotterdam with promising results: the use of the didactic model managed, in some cases, to improve the pass rate of the students, while it

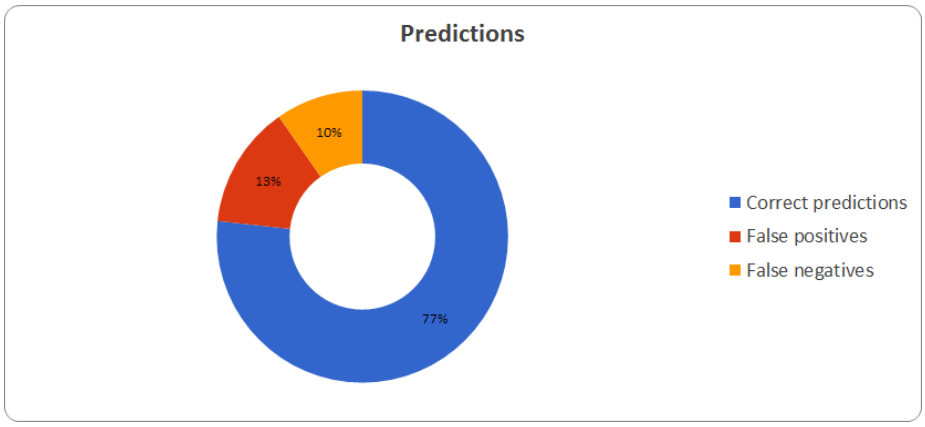

Figure 5: Predition accuracy of GrandeOmega

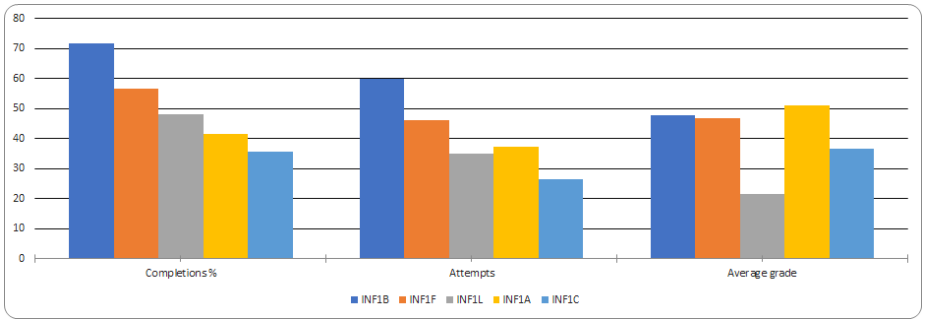

Figure 6: Student performance with and without G.O.

always improved their overall performance (measured as a percentage of the maximum score). Furthermore, the classes that did not use GrandeOmega performed generally more poorly to the point that, in one of them, no students managed to successfully complete any of the assignments. The tool is also able to predict with a reliability of 77% if a student will pass the course on the base of the amount of completed assignments. We can thus conclude that using GrandeOmega boosts the general performance of the students, measured as the percentage of correctly given answers, and, in some cases, improves also the pass rate of the course. On the other hand, students who did not use GrandeOmega at all performed, on average, worse and had a lower passing percentage.

References

[1] Susan Bergin and Ronan Reilly. The influence of motivation and comfort-level on learning to program. In *Proceedings of the PPIG*, volume 17, pages 293–304, 2005.

[2] Department for Business, Innovation & Skills. Experts call for employers and universities to do more to address stem skills shortages. https://www.gov.uk/government/news/ experts-call-for-employers-and-universities-to-do-more-to-address-stem-skills-shortages, 2016.

[3] Facebook. React Javascript library. `https://code.facebook.com/projects/176988925806765/react/`, 2011.

[4] Robert R Fenichel, Joseph Weizenbaum, and Jerome C Yochelson. A program to teach programming. *Communications of the ACM*, 13(3):141–146, 1970.

[5] S. Fincher. What are we doing when we teach programming? In *Frontiers in Education Conference, 1999. FIE '99. 29th Annual*, volume 1, pages 12A4/1–12A4/5 vol.1, Nov 1999.

[6] Mohamed Abbadi Giuseppe Maggiore. GrandeOmega. `http://grandeomega.com/`, 2017.

[7] Irene Govender and Diane Grayson. Learning to program and learning to teach programming: A closer look. *Media 2006 Proceedings*, pages 1687–1693, 2006.

[8] Paul Hudak. *Functional Reactive Programming*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999.

[9] Pattern Insight. Why is up to 45% of software shipped with previously fixed defects? `https://www.theguardian.com/technology/2006/may/25/insideit.guardianweeklytechnologysection`, 2006.

[10] Pattern Insight. Why is up to 45% of software shipped with previously fixed defects? `http://patterninsight.com/why-pattern-insight/`, 2017.

[11] Tony Jenkins. On the difficulty of learning to program. In *Proceedings of the 3rd Annual Conference of the LTSN Centre for Information and Computer Sciences*, volume 4, pages 53–58, 2002.

[12] Ásrún Matthíasdóttir. How to teach programming languages to novice students? lecturing or not. In *International Conference on Computer Systems and Technologies-CompSysTech*, volume 6, pages 15–16, 2006.

[13] Paul Mulholland and Marc Eisenstadt. Using software to teach computer programming: Past, present and future. *Software Visualization: Programming as a Multimedia Experience*, pages 399–408, 1998.

[14] Gordon D Plotkin. A structural approach to operational semantics. *The Journal of Logic and Algebraic Programming*, 60:17 – 139, 2004.

[15] Phit-Huan Tan, Choo-Yee Ting, and Siew-Woei Ling. Learning difficulties in programming courses: Undergraduates' perspective and perception. In *Computer Technology and Development, 2009. ICCTD'09. International Conference on*, volume 1, pages 42–46. IEEE, 2009.