EXPERIMENT AROUND A TRAINING ENGINE — Revision: 1.15

Anne Brygoo
UPMC – UFR d'informatique – Département Recherche
Anne.Brygoo@ufr-info-p6.jussieu.fr

Titou Durand
UPMC – UFR d'informatique – Département Recherche
Titou.Durand@ufr-info-p6.jussieu.fr

Pascal Manoury
UPMC – UFR d'informatique – Laboratoire PPS
Pascal.Manoury@pps.jussieu.fr

Christian Queinnec
UPMC – UFR d'informatique – Laboratoire LIP6
Christian.Queinnec@lip6.fr

Michèle Soria
UPMC – UFR d'informatique – Laboratoire LIP6
Michele.Soria@lip6.fr

Abstract We describe a teaching experiment where an introductory course to Computer Science is accompanied by a computerized training engine. This whole engine relies on the existence of an interpreter of the taught programming language that allows us to offer quizzes as well as exercises with some automatic marking facility. Students may then perform their homework with an immediate feedback without being connected to the Internet. However students' answers are eventually gathered in a central database where they may be analyzed thus providing the means for "personal coaching".

Demonstrations of the engine will be presented at the conference.

Keywords: Software to improve the learning process, Distributed learning system, Improving learning environments

Introduction

Organizing a first introductory course to computer science (CS) raises a number of questions; the first of which being related to the particular programming language to use. This is a hard question that too often triggers holy wars among computer scientists [Bow94] and is far too subtle to be easily justified in front of beginning students. On that question, we think that any answer should take into account the targeted population: the first programming language does not need to be the same if taught to beginning computer scientists or to scientific students who still have to choose where to graduate.

The University "Pierre et Marie Curie" (Paris 6) gave us in 1998 the responsibility of the first introductory course in CS in a cursus named MIAS (for Maths and Informatics Applied to Sciences), a two-year cursus where young students, 18- to 20-year old, study general Maths, Physics, Mechanics and Computer Science before choosing to graduate in only one of these sciences. The CS cursus is made of three other mandatory courses accompanied by an optional CS project. There are roughly 800 students in first year, 600 in second year; 450 get their final degree: 250 choose to pursue mathematical studies, 160 others choose CS.

These figures show that most of our audience were not bound to become computer scientists therefore we decided to introduce the students to the main concept of CS: the "evaluation process" that is, how a computer turns (executes) a text (a program) into some result.

Of course, we also decided that our teaching should attract students to CS. We therefore favored a rather conceptual approach together with numerous programming duties. These ideas are not new and are rooted in the well-known SICP book (Structure and Interpretation of Computer Programs [AS85]) taught for long at MIT.

Teaching is 12-week long and every week is made of one course (1h15') and one lab session (1h30') associated to a pre-lab session (1h30'). Thirty students form a group monitored by one teacher. Every group follows a computer-less pre-lab session. The lab session is performed in specialized class rooms with 15 computers: each computer is operated by two students together.

Last year, we volunteered to prepare an experiment where 50 students would be taught differently. This is this experiment that we describe in the rest of this paper. The main lines of our teaching are detailed in Section 1 as well as the SPAD experiment and how it differs from the regular course. Section 2 rapidly presents the software architecture of the associated computerized environment. The results of the experiment appear in Section 3 followed by some conclusions and future perspectives.

1. Choices, Objectives and Experiment

The goal of the course is to present the "evaluation process" that is, the general principles that allow a computer to interpret a text as a program whose value should be mechanically obtainable. We chose to use a subset of the Scheme programming language for the following reasons.

Our students have a long background in maths, they are used to functions (even if (most often) unary functions from Real to Real) that are the basis of any functional language. Scheme has a very simple and extremely regular syntax allowing students to forget about syntax after a couple of weeks. Excellent books exist such as [AS85, ML95, HKK99] as well as a 50-page long complete and accurate standard [KCR98]. Scheme is an untyped but safe language making it simple and very regular to learn. Scheme makes recursion necessary, unavoidable but still extremely simple. Eventually, Scheme makes simple to write interpreters and is the sole standardized language that can express its own semantics in less than 400 lines.

However even if Scheme is simple, we isolated a subset avoiding notoriously difficult concepts such as anonymous functions, side-effects (no assignment and no mutable data structure) and continuations. Our subset requires to name all internal functions. We adopted the so-called MIT syntax (with internal definitions) that makes program adopt a Pascal-ish look (but without restriction on the types of results). Eventually, our subset allows students to concentrate on the concept of computation as a transformation of information. We also elaborated a "quick reference card" to describe these features and library functions.

Technically, our Scheme subset uses only these linguistic features: variables, functions (definitions and invocations), sequences (`begin`), alternatives (`if`), citations (`quote`) and blocks (`let`). Less than this, we would only be teaching lambda-calculus. However, this set of features is the basis for the majority of mainstream languages.

On the data side, students see homogeneous lists, trees and S-expressions. Since Scheme requires the presence of a GC (garbage collector) no effort is required to manage memory: a disturbing task in most programming languages.

An additional and interesting property of the choice of Scheme is that most (if not all) students are equal. We observe for the last three years that previous exposure to video games, Windows wizardry or imperative scripting is not correlated with good results. Unskilled would-be domineers cannot fake for long: young talents may grow.

Our course is divided into three seasons where the first (six-week long) is devoted to recursion on numbers and lists, the second (four-week long) presents trees and grammars and, of course, recursion on trees and the concept of "abstraction barrier". The third and final season (two-week long) presents the evaluation process as an interpreter for our Scheme subset written in our Scheme subset. The third season does not introduce any new concept. It only gathers many functions (most of which were studied in lab sessions) for a single goal: the evaluation of a small but powerful programming language.

Programming is a kind of literary activity that must be exercised again and again. Moreover a number of habits are adopted more or less unconsciously by studying or even seeing others' programs. Therefore, we strove to provide perfect-looking programs but also imposed precise guidelines to every program written by students. Variables must be appropriately named, functions must be pretty-printed (this is largely automated by the programming environment). Additionally we enforce that any function must be preceded by some structured comments stating its type and specification (this eases a further course using a typed language as well as the interest of abstraction barrier). Any function must also be followed by a predicate testing that very function in a variety of situations: we encourage incremental testing.

Finally, the most difficult points of CS at that level is to master abstraction that is, to differentiate between syntax and semantics, knowledge and information, aspect and meaning. In order to encourage the use of abstraction barrier i.e., interfaces hiding implementation to focus on the behavior only, we encapsulate abstract objects into Scheme values that are printed as small drawings and thus are completely opaque but to the accessors and recognizers of the interface.

1.1. CD-ROM

We made a CD-ROM to support our course. This CD-ROM is targeted for the Windows or Linux Operating Systems. Its aim is to provide to every student a means to practice, at home, the course that is, reading, programming and thinking. Moreover the CD-ROM software provides immediate feed-back wherever possible — without any Internet connection.

The CD-ROM contains the software required to program in Scheme as well as the material for the course including numerous documents related to the Scheme programming language, pragmatics and community. This extra material is provided since a programming language is not reduced to syntax and semantics but also include pragmatics, folklore, programming guides or tricks, etc.

The CD-ROM favors connection-less self-training that is, besides a traditional course (in HTML and PDF form), it offers quizzes as well as exercises that do not require an Internet connection to be performed. More than 85% of our MIAS students have a computer available at home but less than 45% have an Internet connection. Despite this increasing level of wealth and conversely to educational platforms vendors, we strongly believe that, in the next ten years, most students of the world will still not be constantly connected from home to the servers of their University. Therefore we privilege an architecture where students solve quizzes or exercises, submit their answer and have an immediate feedback telling them how good that solution is: the feedback is computed locally and does not require an active connection.

The current version of the CD-ROM contains nearly 400 questions of quizzes and 245 questions within 58 exercises. This gives great latitude to students (and teachers) to choose which exercise to practice (or study).

1.2. The SPAD experiment

The University decided, a year ago, to experiment with distance learning at the MIAS level. In September 2001, a group of nearly 50 students began to be taught in a new way in Maths and CS for one semester.

First, the students only needed to spend three days (instead of five) at the University. Second, every student received a CD-ROM containing the computerized teaching material. We decided, in CS, to organize the students' week on a new basis: we suppress the course and merge the pre-lab and lab session into a single weekly "pedagogical rendez-vous" (2 hours). We also reduced the size of a group to 15 students so every student may practice alone on a computer.

However since we wanted to follow the progress of our SPAD students, the CD-ROM software locally stores all their answers and emits them, later on, to the servers of the University when an Internet connection is on: these are the "traces" as can be seen on Figure 1. Most often, with a few days lag, we get a precise view of our groups enabling us to advise our students on a personal basis via mail or a shared forum.

The same content is delivered to the regular students of MIAS as well as to the students of the SPAD experiment. A similar chronology was adopted to ease students shifting from SPAD to regular (actually we observe the contrary). SPAD and regular students both have the same final examinations.

Granted these constraints, we organize the week as follows:

1 Tuesday was the day of the pedagogical rendez-vous. Week assignments (course, quizzes and exercises) were prescribed.

2 The course should be read and first-level quizzes should be done for Friday.

3 Questions about the course should be posted (on a forum) by Sunday

4 Exercises should be made by Monday.

The organization of the week rules the content of the pedagogical rendez-vous. We answer student's questions, we shortly present the most delicate point of the written course and, finally, we supervise the students practicing quizzes or exercises as in a normal lab session.

2. Training engine

The CD-ROM contains — a PDF version of the written course so it may be searched or printed, as well as — an HTML version chopped into pages centered on a single topic. This course is intended to be the main document containing all sort of links to quizzes, exercises or other pages with extra information.

Since the course heavily uses a programming language, we also provide a programming environment for that language: we chose DrScheme by the PLT team from Rice University [FCF+01]. DrScheme is a nice environment with a lot of well-thought pedagogical features. It runs on a variety of Mac, Windows and Linux boxes and is easily installed on all sorts of computers.

Quizzes and exercises are written in Scheme, they are installed as an additional package to DrScheme where they run as separate threads. As we regularly improve and extend this package, students get used to update their configuration (with a simple click).

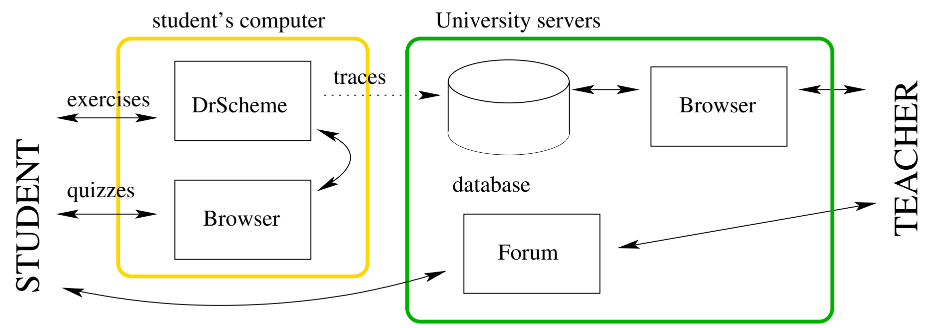

Figure 1. Overview of the architecture

2.1. Quizzes

Quizzes help students to understand the course before getting involved in exercises. Quizzes adhere to the structure of the course and provide questions on every topic of the course. Quizzes are ranked from easy to difficult and from optional to mandatory. Students are aware of this ranking. Quizzes are not marked, answers are just checked for correctness wherever possible.

After reading a topic of the written course, the student is proposed some quizzes as simple links. Technically, clicking on such a link directs the browser towards a web-server embedded within DrScheme (cf. Figure 1). This web-server loads the required quiz (a Scheme file) and starts evaluating it. This program (the quiz) is made of a succession of standardized questions exerting various abilities of the student [Que00]. We distinguish three levels of quizzes:

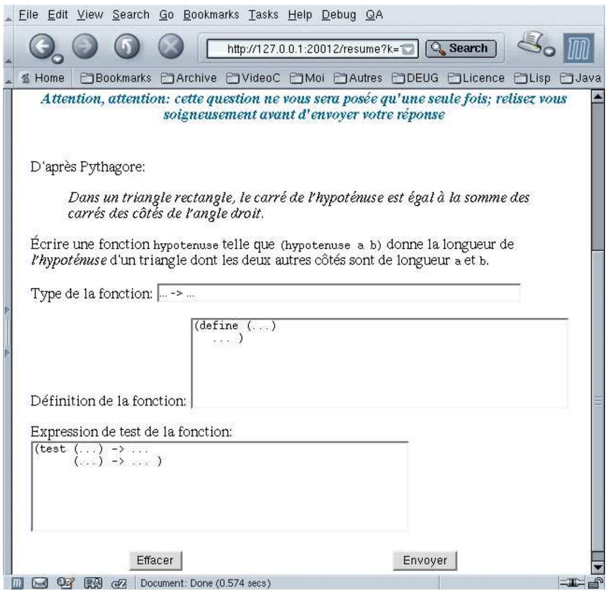

Figure 2. Screen capture of a quiz – This quizz corresponds to a compound question where the student has to write a function that is, its type, its definition and some associated tests.

1 Simple applications of the course. These questions verify that the student knows how to solve some simple problems. The answers to these questions are checked to be correct. (More on this below.)

2 Questions on the course itself, where the student is invited to write his understanding on particular points. These questions help the student to check whether he understands the proposed points. These questions are not formally checked but hypertext links to the paragraphs where the question is handled in the course are offered as response.

3 "Meta-questions" that replace the knowledge in the perspective of the whole course. Their aim is to verify that the student understands how the acquired knowledge contributes to the overall goal of the course.

The standardization of questions makes easier for students to recognize the type of question they have to solve. It also makes easier for teachers to write quizzes since only the varying parts are to be specified. The HTML decoration is therefore totally unrelated to the scientific content. Among the types of questions are:

1 What is the value of that excerpt ?

2 Write a program that gives that value ?

3 What is the type of that function ?

4 Provide a context where to embed that excerpt and give the expected value ?

5 Define a function that fulfills that specification ? etc.

To sum up, most of the quizzes allow students to program short things in Scheme, without the complete DrScheme programming environment, with the sole power of a browser.

2.2. Exercises

An extra menu item within the DrScheme programming environment allows students to choose an exercise, cf. Figure 1. Exercises are made of a series of questions. Each question is displayed in HTML in a separate window. A question asks for the definition and the test of one (or more) Scheme function(s). First, the student writes the required function followed by some tests, he may then hit the "Check" (In French "Tester" as in Figure 3) button to get some feedback for his work. As developped below, the feedback consists in a mark associated to some comments justifying this mark. The marking process takes into account many syntactical or semantical aspects of the program but is not intended to replace the teacher. The mark is an indicator that tells the student how correct is his program. Additionally, if the mark is over some threshold, an (HTML) solution is displayed.

Let us say that a function named f is asked for, then the student has to write such a function as well as some tests gathered in a predicate named `test-`f, this predicate should return True if f satisfies all the tests. Marking proceeds as follows:

1 Does f and `test-`f exist ?

2 Does f satisfy `test-`f ? A student solution is required to be coherent.

3 For any solution of the teachers, say f_T, does f_T satisfy `test-`f ? This checks that student tests are not biased towards his solution.

4 Does f satisfy teachers' checking function `test-`f_T ? This checks whether the student function is correct viz our tests.

Functions f_T and `test-`f_T are provided in compiled form in the library context of the exercise. Students cannot use them but the definition of f_T appears with some comments in the HTML solution page.

Besides this behavioral verifications, some syntactic checks are also performed to verify that students functions respect the taught subset of Scheme. We envision next year to improve these verifications:

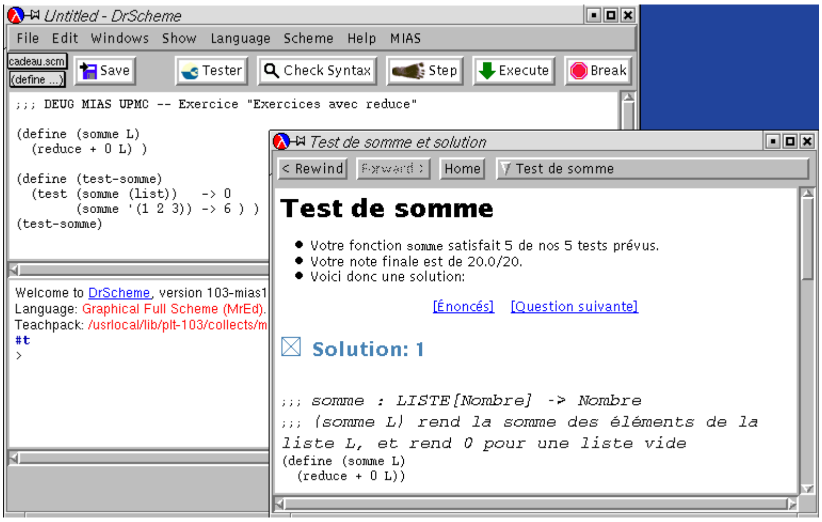

Figure 3. Screen capture of an exercise – The student hit the “Tester” (check) button and got a mark good enough to let him see a solution (more than one appears here not seen in this screenshot).

- We may instrument the interpreter to compare the complexity (number of calls, number of allocations, etc.) of the student’s solution with ours.
- We may check the type of the definition with respect to the associated specification. Soft typing is a convenient choice for Scheme [WC97].
- Finally, we may perform a test coverage analysis to analyze how the student tests its function (and our solutions).

2.3. Traces

For all quizzes and exercises, solutions and their evaluations (most often a number) are time-stamped and stored (more or less immediately) in a database of a central server of the University, see Figure 1. We developped, for our own usage, some SQL web-based forms inquiring the database to display the state of any particular student with respect to the quizzes or exercises of any given week thus allowing us to make some comments (by mail) on his answers: this is what we call “personal coaching”. We also inquire the database to display the state of a whole group with respect to the quizzes or exercises of any given week to allow us to write a page entitled “Week advices” or a post in the forum where we comment upon a popular mistake or habit.

These forms also allows some statistical analyzes to determine which questions (quiz or exercise) have a high failure rate either because the question is poorly worded or infeasible at that place in the course.

3. Results and perspectives

Seventy students were at first volunteers for the SPAD experiment. Some of them withdraw when they realize that the CD-ROM will also be available for regular students. At last, around forty students were enrolled, they attracted 10 more students by spreading the word. The availability of a computer and of an Internet connection were the only pre-requisites to volunteer.

A first and surprising result is that the SPAD group is rather representative of the whole group of regular students: we feared a group of male-only CS-addicts and took great care in the composition of the SPAD group. However, after the initial selection, we observe students that are always absent, students loosely interested in CS, students that work regularly with poor results, students with a huge background on Windows-based software but unable to master recursion, students without any former programming experience but with good results.

The continuous use of a computer for that course showed unanticipated effects. SPAD students use the computer to read the course, follow the links towards quizzes, perform quizzes on screen, switch between the browser and the programming environment. Moreover, they are alone on a computer during lab session therefore, they are much more at ease with computers than the rest of the MIAS students: the computer became an helper device rather than an opponent to be tamed. SPAD students use the computer as a specialized co-worker.

Taking this habit into account, we urge our SPAD students to realize that, since the examinations are on paper only, they will not have any computer available to help them to test their solutions during examinations. They seem to surmount that problem.

Figure 4 shows the final results for both populations (the dark plot represents the SPAD students, and the light one is for regular students). We plotted the percentage of students with a given mark.

These statistics concern about 750 regular students and 50 SPAD. The proportion of students with mark zero (essentially meaning "absent") is 24% among SPAD students, and 17% among the others (this is the usual ratio of students leaving during their first three months at the university). This difference can be explained by the fact that 10% of the students that chose to be SPAD are suffering from severe illness, and could not complete the course. Except from this initial difference, the two graphs look very similar : normally distributed, with mean value around 9 (for SPAD students) or 10 (for regular students), and with a non negligeable proportion of very good students (around 15% with a mark greater than 15 over 20).

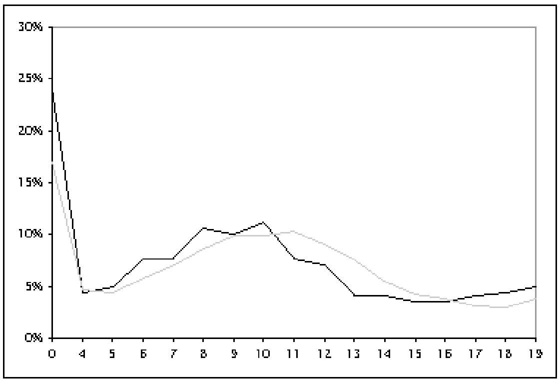

Figure 4. Final results for SPAD students (dark plot) and regular students (light plot)

Besides these raw results, we also want to report a refined perception of the experiment, based upon the final questionnaire that was fullfilled by all students, and several discussions with the SPAD students. We shall emphasize below on quizzes and exercises, but let us just now claim that the SPAD students did enjoy the experience : the main points that were brought out were

- they feel free to organize their work, but very much appreciate the weekly precriptions they are given (see 1.2).
- though they meet their teachers only once a week (and would rather have two rendez-vous), they feel closely connected with them via mail and the daily maintained forum.
- they appreciate learning programming by practicing (even for those who had no experience with a computer, and first had to struggle a lot).

3.1. Quizzes

Quizzes were greatly appreciated. They are considered as fun compared to other scientific courses because of the rich variety of questions: instead of a finite number of boxes to check or click, we offer questions with a far wider spectrum of answers. For instance, when we ask for a program whose value is `2/3`, one may answer `(/ 2 3)` or `(* 2 (/ 3))` or even `(let ((two '(I I))) (/ (length two) (length (cons 'I two))))`. The mere existence of an `eval` function is the key of that feature.

A question within a quiz is not reduced to the terms of a problem followed by a set of answers, some of them flagged as being correct. We see a question as the terms of a problem accompanied by a predicate telling if an answer (a text or a click or some clicks) is correct. The answer is interpreted (in a confined environment) then checked against the specification of the terms.

The availability of an interpreter allows any course to have similar quizzes. Courses on programming language may obviously benefit from this feature. The interest of Scheme is that first, the `eval` function (that represents the interpreter itself) is predefined in Scheme [KCR98], second, that to write variants of such a function is well understood for many years [SS78, Que96].

Each question of a quiz may be performed rather quickly contrarily to exercises that require a much greater time. However since quiz files may chain a number of questions (between 5 and 15), students must perform them in one go; quizzes cannot (currently) be suspended and later on resumed. We intend to analyze the statistical results of quizzes in order to provide either automatic shortcuts for skillful students or additional questions where needed for unsuccessful students.

Not only quizzes prepare to exercises but, as students asked us, they may also be used to prepare the final examination. They consider the individual questions forming quizzes as a big database from which may be dynamically extracted new quizzes on selection-able themes. This would allow students to focus on some precise topics. This improvement requires some meta-data to classify, indexize and rank the difficulty of all questions of all quizzes. This improvement was also asked by colleagues who want to create new arrangements.

Being programs, quizzes are displayed in HTML, page after page. This is nice but not appropriate when commuting from home to the University or back. Students and colleagues asked us for paper version of quizzes. We provided such a view for questions only. Being programs, answers cannot be displayed since they are represented by predicates checking the appropriateness of students' answers.

3.2. Exercises

Exercises are longer, they take a quarter of an hour to several hours to be completed. The availability of a "Check" (in French "Tester") button that analyzes the current answer and return a numeric mark was very well appreciated. It allows students to have an immediate feedback on their progression in absence of any teacher. We also use exercises in lab sessions where it allows students to proceed at their own speed.

Exercises are marked on their behavior. We intend next year to also measure the complexity of the solution compared to our solutions (number of calls, use of recursion, number of access in memory, etc.) in order to give better advices. While in lab, we regularly check the style of their solutions and make some comments to improve their writing. We do the same when looking at the traces and mail back some advices.

However we observe two interesting deviations. Solutions are displayed when the mark is over some threshold. We observe that they are rarely studied as they deserve. We envision to improve our exercises to force the use of these solutions and some of their properties in further questions. Currently a student may only see a solution while working on the same exercise (solution pages are crypted on the CD-ROM to avoid cheating). We envision to record (persistently) the right to see that solution so students may return to that page and study it later on. Next year, the CD-ROM will allow that a right acquired in lab may be used at home or vice-versa.

Hitting a button to get a mark is easier than evaluating the program, checking that the tests are indeed correct and so forth. Even us as teachers when writing exercises tend to favor that mode! Fortunately, since the marking process checks a number of features, this is safe. Only in case of errors yielding a bad mark, students check manually which test is not right and why. However any small mistake that yields a mark above the threshold is, usually, completely ignored.

We used quizzes and exercises as the infrastructure for small marked examinations during lab sessions; students may use their programming environment to write their answer. During the examination, we were observe that half of the students write their test functions! We were surprised since this was not required by the examination. To encourage that behavior, we required these test functions starting with the next examination.

4. Related work

We share with ELM-ART, an intelligent tutoring system on WWW to teach Lisp, [BSW96, WS97] a number of goals and means. We teach a similar language (Scheme is an heir of Lisp) and our web-server (for quiz) is written in Scheme (whereas ELM-ART uses CL-HTTP written in COMMON LISP). There are many differences though (see Figure 1). Our system uses primarily the programming environment that is, DrScheme. This allows students to write Scheme programs with great comfort, this is not the case with any Web-based system we know of. It also allows more interesting exercises where we provide some libraries to be used by the students. Students have access to all the debugging means provided by the programming environment to perform their assignment.

Traces are emitted by the programming environment and are asynchronously accumulated into a central database. Various web-based SQL-request forms extract students records from that database thus allowing teachers to advice students by mail or by voice at the weekly pedagogical rendez-vous. Since we now have a database full of more than ten thousands answers to the various quizzes and exercises we provided, we intend to study then statistically in order to elaborate a taxonomy of these answers and to partially automate the synthesis of advices.

This is a very asynchronous feedback compared to the ELM-ART solution that is instantaneous but requires students to be always online, a situation we explicitly avoided. Instantaneity was not a major concern since we must keep our SPAD students synchronized with the non-SPAD students and we are bound to a constraining week organization.

The way we mark exercises by comparison to teachers' solution is very easy to put to work, this solves one of the major difficulties highlighted in many works [JCL00] which is to write these marking functions. The work is reduced to write at least one solution then to decide on which (possibly dynamically-generated) set of inputs, the solution and the student's answer should be compared. Given that we use Scheme and run our tests on the values themselves (i.e., their representation in memory) instead of their printed representation, we are free from the burden of specifying any precise IO format: this is quite similar to the Boss2 solution [JCL00] that uses Java interfaces to hide from implementation details.

5. Conclusions

As is the case for nearly all teaching experiment, we feel that the experiment is a success both for students who liked the freedom they gained with this organisation and for teachers who experienced new means of teaching.

The material we developped for the SPAD experiment is not only useful for SPAD students, it brings the oppportunity for all our MIAS students (and possibly other French-speaking students around the world) to practice at home, with feedback, the quizzes and exercises supporting our course.

Moreover, we gathered a number of pedagogical resources offering new teaching possibilities. Computerized homework with automatic submission (via the trace system), computerized examination during lab sessions, computerized revision (with quizzes) during free-hours. We are eager to explore these new fields with our colleagues.

Eventually, we collected a number of traces that we plan to data mine. Much elaborated studies may take place, for instance computing the difference between two consecutive answers to a same question. This will allow to understand better the learning process in order to improve our teaching.

The web site we developped for this course is freely accessible (but mostly in French) at:

`http://www.infop6.jussieu.fr/deug/2001/mias/mias-a/deugspad/`

References

[AS85] Harold Abelson and Gerald Jay Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Mass., 1985.

[Bow94] H Bowman. A perspective on language wars. In *Papers for CTI Annual Conference 1994*, 1994.

[BSW96] Peter Brusilovsky, Elmar W. Schwarz, and Gerhard Weber. ELM-ART: An intelligent tutoring system on world wide web. In C. Frasson, G. Gauthier, and A. Lesgold, editors, *Intelligent Tutoring Systems (ITS'96)*, volume 1086 of *Lecture Notes in Computer Science*, pages 261–269. Springer-Verlag, 1996.

[FCF$^+$01] R. Findler, J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler, and M. Felleisen. Drscheme: A programming environment for scheme. *Journal of Functional Programming*, 2001.

[HKK99] Max Hailperin, Barbara Kaiser, and Karl Knight. *Concrete Abstractions, An Introduction to Computer Science Using Scheme*. Brooks/Cole Publishing Company, a division of International Thomson Publishing Inc, 1999. ISBN 0-534-95211-9.

[JCL00] M.S. Joy, P.-S. Chan, and M. Luck. Networked submission and assessment. In *Proceedings of the 1st Annual Conference of the LTSN Centre for Information and Computer Sciences, LTSN-ICS, 2000*, 2000.

[KCR98] Richard Kelsey, William Clinger, and Jonathan Rees, editors. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998. Also appears in ACM SIGPLAN Notices 33(9), September 1998.

[ML95] Vincent Manis and James J Little. *The Schematics of computation*. Prentice-Hall, 1995.

[Que96] Christian Queinnec. *Lisp in Small Pieces*. Cambridge University Press, 1996.

[Que00] Christian Queinnec. The influence of browsers on evaluators or, continuations to program web servers. In *ICFP '2000 – International Conference on Functional Programming*, pages 23–33, Montreal (Canada), September 2000.

[SS78] Guy Lewis Steele Jr. and Gerald Jay Sussman. The art of the interpreter, or the modularity complex (parts zero, one, and two). MIT AI Memo 453, Massachusetts Institute of Technology, Cambridge, Mass., May 1978.

[WC97] Andrew K. Wright and Robert Cartwright. A practical soft type system for scheme. *ACM Transactions on Programming Languages and Systems*, 19(1):87–152, January 1997.

[WS97] Gerhard Weber and Marcus Specht. User modeling and adaptive navigation support in www-based tutoring systems. In Anthony Jameson, Cécile Paris, and Carlo Tasso, editors, *Proceedings of the Sixth International Conference on User Modeling (UM'97)*, pages 289–300, Cagliari (Italy), June 1997.