Olympiads in Informatics, 2008, Vol. 2, 149–162

© 2008 *Institute of Mathematics and Informatics, Vilnius*

Early Introduction of Competitive Programming

Pedro RIBEIRO
Departamento de Ciência de Computadores, Faculdade de Ciências, Universidade do Porto
Rua do Campo Alegre, 1021/1055, 4169-007 PORTO, Portugal
e-mail: pribeiro@dcc.fc.up.pt

Pedro GUERREIRO
Universidade do Algarve
Gambelas, 8005-139 FARO, Portugal
e-mail: pjguerreiro@ualg.pt

Abstract. Those who enjoy programming enjoy programming competitions, either as contestants or as coaches. Often coaches are teachers, who, aiming at better results in the future, would like to have more and more students participating, from earlier stages. Learning all the basic algorithms takes some time, of course; on the other hand, competition environments can be introduced right from the beginning as a pedagogical tool. If handled correctly, this can be very effective in helping to reach the goals of the course and, as a side-effect, in bringing larger crowds of students into the programming competition arena.

Key words: programming contests, computer science education, automatic evaluation systems, competitive programming, introductory programming, IOI, International Olympiads in Informatics.

1. Introduction

The most popular programming competitions are geared to imperative languages and to input-output problems. The automatics judges that handle submissions have been customarily designed for that environment. In our own programming courses, at our universities, we are using one such judge as a learning tool. Students submit their programming assignments to the judge, with the benefit of immediate feedback, more reliable than what could be provided by a teaching assistant. Moreover, the automatic judge publishes a ranking of the students by number of accepted submissions, and this induces a healthy competitive attitude that many students enjoy and that makes them work harder and in a more disciplined way than they would otherwise.

This approach has shown to be very effective in increasing student productivity, measured by the number of programs actually written by the students during the courses, but it is not yet completely satisfactory. In fact, one cannot use the customary approach of the judge before students have learned how to do input and output. In some languages (C and Java, for example) this is not a trivial task, and it is never a trivial task when large sets of data have to be handled. Furthermore, it makes little sense to spend time on the details of formatting numbers or splitting strings when problem solving using computers is

barely being introduced. In any case, it is a pity not to use the automatic judge to validate the very first pieces of code with which the novice programmers are struggling, because they are incomplete programs, and because they do no input-output. It is a pity because there is a large number of small tasks that the students could try out and which could not reasonably be checked by a human with the promptitude that we seek, and because we have observed that students derive a special pleasure when the results of their efforts are immediately rewarded with an "accepted" message from the judge.

On the other hand, there is a trend towards using functional languages such as Haskell in introductory programming courses. These languages run on an interactive "calculator", in which functions are called with supplied arguments and results are displayed automatically. The same situation occurs in logic programming, with Prolog. In these environments there is no input-output, in the conventional imperative programming sense, with reads and writes, at least in the initial stages. This means that functional programming and logic programming can be used with the automatic judge, from the very beginning, since the input and output are handled automatically, so to speak. This has completely modified the first labs: the initial exercises are now online, meaning they are to be submitted to the judge for feedback, whereas in the past students would pass to the next exercise after "manually testing" once or twice only. In addition, the competitive attitudes that we observed made most students want to solve all the exercises, to get more points, and be in the top positions of the informal rankings.

Indeed, the lessons learned from approaching programming using these "unconventional" languages can be brought back to the imperative style of programming. With Pascal, C, C++ or Java, we can use the judge in an "advanced" way, similar to the one used for Haskell or Prolog, with incomplete programs that do no input or output, even if this requires some hacking to overcome its original design of the judge. With this, we have been able to provide to our students, in various courses, an early introduction to competitive programming.

This paper is organized as follows. After this introduction, we briefly describe, in Section 2, the competition environment that we use in our programming courses, which is based on the automatic judge Mooshak (Leal and Silva, 2003; Mooshak site). Then we discuss how we have tweaked it into being able to use problems with no input-output in conventional terms, first with Haskell, in Section 3, then in Prolog, in Section 4. In Section 5, we bring those techniques to conventional programming languages, allowing novice students, even those who are programming for the first time, to immediately have their small, incomplete, programs automatically checked. In Section 6, we report on the feedback we had from students who were exposed to this approach in the introductory programming course with Haskell. A conclusion follows.

2. Competition Environment

There are several systems for automatically assessing programming assignments (Douce *et al.*, 2005; Ala-Mutka, 2005). Each one has strengths and weaknesses. Given our own

background, it was only natural that we chose for our courses an automatic evaluation system related to programming contests: Mooshak. We have been using Mooshak to manage the programming competitions that we organize for more than five years, and we know exactly how to take advantage of its strong points, avoid the dark corners, and even extend it to do things it wasn't originally designed for. In all of our courses, these days, we make extensive use of this publicly available platform.

Mooshak grew from previous experience with a late 90's web-based learning environment for Computer Science called Ganesh (Ganesh Site). Mooshak appeared first on the context of the ACM ICPC (ICPC Site). In this family of contests, typically you have teams of three students solving problems in C, C++ or Java. Solving means creating a program that will compile under some given compilation flags and that, when it runs, produces the correct output for the secret input files associated with the problem. Mooshak was first published in 2000 and since then it has matured into a very robust, flexible and accurate platform. In its initial realm of ICPC, it has been used successfully in several editions of local, national and international contests, such as the Southwestern Europe Regional ACM Programming Contest (*SWERC 2007* site).

Since the beginning, Mooshak architecture was designed having in mind different kinds of programming competitions. For example, in ICPC a program is considered correct only if it passes all tests; otherwise it does not count. In the Olympiads, there is a partial credit scheme and points are allocated depending on which tests passed; i.e., a contestant will get points even if his program fails some tests. Mooshak provides a way to give a value to each test, and ranks the contestants accordingly. Moreover, it has way of creating customized ranked schemes. So, one could for example grade the submissions according to the source code size, or to the memory usage. Finally, it is possible to run in a mode that permanently displays the current ranking, useful for ICPC, or that hides it, useful for the Olympiads.

In terms of feedback, Mooshak is more complete than other automatic judges. Instead of just giving the result of a submission as a crude "Wrong Answer", "Runtime Error" or "Compile Time Error", Mooshak allows us to define more detailed reports that really give the students more opportunities to understand what went wrong. This is especially important with compilation errors. In an on-site contest, all contestants are using the same environment as the judge, but on a distributed contest, or in a classroom environment, it is common that student use heterogeneous platforms. Even if very strict compilation flags are used, to ensure adherence to the language standard, it is not uncommon for a program that compiles successfully at the student's computer to fail in the judge. In this case, Mooshak can present the compiler error message, guiding the student in solving the issue. This, by the way, has the nice side effect of leading students to better appreciate the advantages of following standards. In the case of runtime errors, the system can be set to provide the submitter with the exact kind of exception that the program rose. This can be important from an educational point of view, because we don't want the students to be frustrated for committing mistakes that are common to beginners, but instead we want them to learn from errors, helping to achieve correct solutions.

Mooshak's user interface is completely web-based. It has four basic profiles: contestant, judge, administrator and guest. For contestants (or students, as in our case), the

interface shows the problem descriptions, and allows them to submit their code and ask questions. The judges (the teachers) can go through the questions and answer them using the platform. Thus, everyone will see all the questions and all the answers, in an organized way that also fosters interaction between students and teachers. The system also has the capability of issuing global warnings, and this is very useful in an "emergency". Judges have access to a detailed report of every submission, and can check in detail what result was obtained in each test case. If necessary, they can even manually change the result of a submission or ask for an automatic re-evaluation (for example, in case the tests are modified after the contest started). The administrator profile is used to set up a contest and to upload all files concerning it. Finally, the guest profile is available to the public and does not require authentication. With a guest login, anyone can take a look at rankings and at the submissions as they are being made. Indeed, we have observed students proudly giving the URL of their course's Mooshak site to friends, in order to show off their achievements. An example screenshot of the contestants' view can be seen on Fig. 1.

Perhaps the strongest point of Mooshak is its simple yet very flexible way of evaluating programs. Basically after some code is submitted it passes through three phases. The first one is compilation. Compilation is completely customizable in the sense that the administrator can set up the command line that calls the compiler. Normally it is used precisely for compiling the source code and obtaining an executable, but nothing forbids us from doing whatever else we want in this phase. Mooshak considers a compilation to be successful if the program (or programs) run in the compilation phase do not write anything to the standard output. In case the compiler does output something and we want to ignore it, there is a mechanism for that, based on regular expressions. Although the compilation phase is meant for compiling, we have been using it for other purposes as well. For example, in some occasions, we use Mooshak for accepting other types of as-

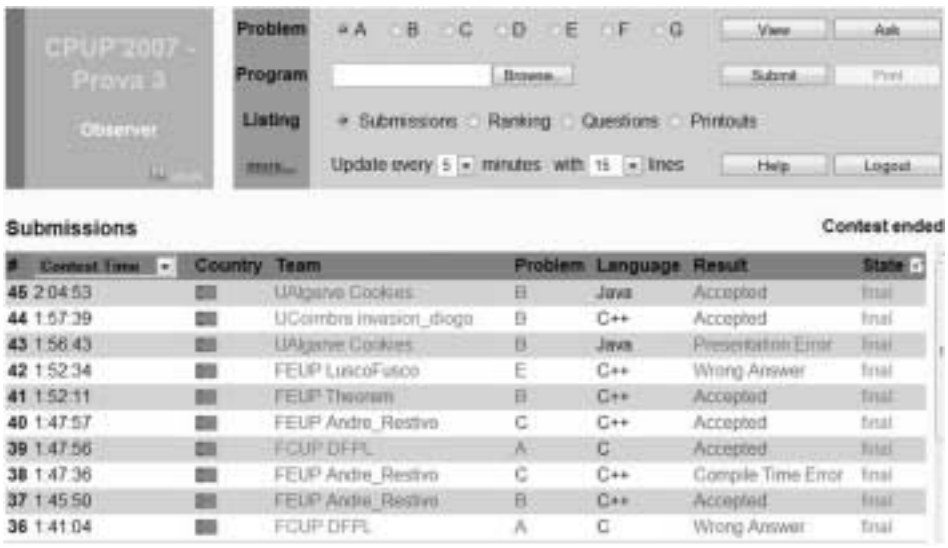

Fig. 1. Example screenshot of Mooshak in contestant mode.

signment, and we use the compilation phase to check whether the submitted zipped file contains all the specified deliverables, in the required format.

The second phase is execution. The program is run once for each test case, again by way of a customizable command line. Mooshak provides a safe sandbox for execution of programs, so we do not have to worry about the students trying to do anything nasty with their code. We can create boundaries for the sandbox, specifying things like the maximum CPU time allowed or the memory limit. For each test case, Mooshak feeds the command line with the content of corresponding input file and it stores the output in a temporary file.

The third phase is the evaluation itself. Mooshak provides a default evaluator which compares the output that was obtained with the expected output. For flexibility, it also allows us to define a custom command to be applied to the output file, thus evaluating it directly.

With this simple versatile scheme, Mooshak can be tuned to a variety of programming languages. Indeed, it has been successfully and routinely used with the standard competition languages – C, C++, Java and Pascal – but also with other imperative languages such as Perl and Python. Moreover, it is able to cope with functional languages such as Haskell and logical languages such as Prolog, as we will see in more detail. Colleagues are also using it for evaluating shell programs.

Although Mooshak was initially developed for a particular kind of programming contests, it was designed in such a flexible way that it can be used for a multitude of things. Basically, we can think of it as a platform that provides a sandbox for safe execution of user code, together with an interface for submitting that code and a user management system. In what concerns us here, this set of facilities also makes it a very fine pedagogical tool.

3. Using Competitive Programming with Haskell

Over the years, we have been using an imperative-programming first approach to teaching programming, at university level (CC2001). More recently, we introduced the automatic judge Mooshak as a pedagogical tool, and this has proven to be invaluable. With it, not only the amount of programming done by the students increased several times, but the courses themselves became more joyful, and also more popular, more transparent and more spoken of. It also made students more diligent programmers, because deadlines, being enforced automatically, are more difficult to circumvent, and more rigorous programmers, because of the greater effort required perfecting the programs so that they pass all the secret tests held by Mooshak.

The first contact of students with Mooshak tends to be turbulent. Usually students react angrily when Mooshak refuses to accept their solutions, which they could swear were bulletproof. Moreover, at the beginning of the semester, good students are striving for visibility in the class. It must be very disappointing to have your programs rejected and not being able to hide it.

Indeed, by having Mooshak evaluate students' programs, we are creating a new obstacle to students: they must solve the programming exercise and they must do it a way that Mooshak accepts. Take, for example, the problem of solving a second degree equation. The problem is not difficult to understand, but it has its own small programming complications: the three coefficients must be read, the roots computed, not forgetting that the equation may have no real roots, and then the roots must be written, if they exist, or else some message must be produced. Since we are handling real numbers, for automatic evaluation to be possible, we must specify precisely the number of digits after the decimal point, which root to be written first and how to separate the two roots.

Most likely, as teachers, we would like our students to concentrate on the function that solves the equation. However, students are eager to see the results, and if they are still insecure about functions, they will try to do everything – reading, solving, writing – in the main program.

Conventional input-output automatic judging does not help here: it only cares about the input-output behavior. In a way, automatic judging somehow works against us, teachers, in the elementary stages, from that point of view. On the other hand, it makes life harder for the students, unnecessarily, by requiring them to master the details of input of variables and output of expressions, which we might want to downplay initially.

More recently, we adopted the functional-first approach to programming, using Haskell. Of course we wanted to continue using automatic judging, with Mooshak, but for that we knew we would have problems on two fronts. On the one hand, Mooshak was designed for compiled languages. We would be using Haskell with an interpreter in the course and we would want Mooshak to rely on that interpreter, to avoid superfluous mismatches. On the other hand, input-output in Haskell is an advanced topic, one that typically is not addressed until the final weeks of the course. It is not necessary initially because functions are called in the interpreter, acting as an interactive calculator, with arguments typed in as needed, and results are written back, automatically.

In order to be able to run autonomously, Haskell programs must have a main function. However, unlike C, for example, we can live without the main function for a long time. Our programs are simply collections of functions and we can call any of them, from the interpreter, providing the arguments each time. This is fine, from a pedagogical perspective, because it impels students to organize their solutions as collections of functions, instead of long lists of intricate instructions within the sole main function. It also frees them from the worries of input, since the values for testing are typed in as arguments of the function, and of output, since the result of the function is written automatically, using a default format. Again, this clear separation between computation and input-output can have a very positive influence in the early habits of the students.

For Mooshak, a compilation is successful if there are no error messages on the standard output. The interactive interpreter is not useful for this. Instead, we used a standalone version that loads the source file and runs its main function. Since the file submitted by the students does not have a main function, Mooshak adds one, via an ad-hoc script, just for the sake of compilation. This main function is empty. Therefore, if the submitted file was syntactically correct, the program does run in the interpreter and produces no

output, just like a successful compilation. If something is wrong, the stand-alone interpreter outputs a message, and that is enough for Mooshak to understand that there was an error.

After the submitted program passes the compilation stage, the function to be tested must be called. Typically, we want to call it several times, with different arguments. Recall that normal Haskell functions, the ones we are considering in this discussion, perform no input-output. So, instead of using an input file from where the data is read, we use a Haskell source file with a main function calling the function to be tested, using the desired arguments. Indeed, for flexibility and simplicity, this file replaces the input file that is provided for each test in the standard operation of Mooshak. Our source file is appended to the submitted program, via a script, and the result is run by the stand-alone interpreter. The result of each function call is written on the standard output and can be compared to the expected output.

By hacking Mooshak this way we were able to use it from the first day of our Haskell course. At the beginning, the programming exercises dealt with simple functions, such as counting the odd numbers in a list, checking if a given number appears in a list, etc. Students first write the programs using a text editor and experiment on their own with the interactive interpreter and then submit to Mooshak. For each accepted submission, students earn one point. For these simple programs, it seemed easy to have an "accepted" and that helped students gain confidence in the system. This contrasts with our experience in previous courses, where, on the contrary, most early submissions were erroneous, which left students very uneasy.

We used this setup for the most part of the course, not only for exercises involving simple functions, but also for larger assignments and for programming problems similar to those used in competitions. Only in the final part did we introduce input-output in Haskell, and we came back to a more conventional use of Mooshak, with input and output handled explicitly by the program and a main function that is called by the interpreter, by default. At this stage, students were more capable of overcoming the details of input-output than they could have been in the beginning. However, the presence of the main function requires a change in the compilation script, because we do not want the program to actually run in the interpreter in the compilation phase, which is what would happen because the provided main function was not empty, as the one we added before. Well, the new script edits the source file, replaces the identifier main by another unlikely identifier and then passes it on to the old script.

Overall, in this course, there were 69 problems to be submitted and evaluated by Mooshak. This is much more than could have been handled manually by teaching assistants. Not all students had all their submissions accepted, of course, but all of them certainly got a lengthy exposure to the automatic judge. The points earned by the accepted submissions counted for 30% of the final grade.

Although we did not stress the competitive aspect of the assignments, many students thrived in it. We observed that many of them enjoyed being on the first places on the ranking and made an effort to solve the problems as soon as possible. As anecdotal evidence on this, at one occasion, we made a mistake when setting up a new contest in Mooshak

which caused the ranking to disappear. We immediately received complaints from the students that "without the ranking, it was no fun".

As a side effect of their involvement with an automatic judge designed for managing programming competitions, some of these students developed a liking for this style of competitive programming. Perhaps some will participate in competitions in the future. However, most programming competitions still use only the usual programming languages – C/C++, Pascal and Java – and these students are not prepared for them just yet.

4. Using Competitive Programming with Prolog

Very much in the way described in the previous section, we were able to use Mooshak in a course with the main focus on logical programming, using Prolog. Instead of just testing "by hand" their Prolog programs, students use Mooshak for that. At the same time, teachers obtained a detailed picture of the successes and failures of the students and part of this information was used for grading.

Like in the case of functional programs, we do not usually compile logic programs; instead, we load them on an interpreter, in our case Yap (Yap site), a fast Prolog interpreter developed also at Universidade do Porto. Contrary to imperative languages, in Prolog we declaratively express a program using *relations*, called *predicates* in Prolog, in a way similar to databases. Executing a Prolog program is actually running a *query* over those predicates, and typically this is done using the command line of the interpreter. We test by making queries about the relevant predicates. The Prolog environment then uses inference rules to produce answers.

A Prolog program therefore does not need a "main function": it is simply a collection of predicates. In the context of Mooshak, compiling can be seen as loading the program in the database. Executing a program is done by running a series of queries. We use a metapredicate that collects the list of all answers that satisfy the goal, comparing that list with the expected list of solutions (ignoring the order). Often, the solution contains only one possible answer (for example, the length of $[a, b, c]$ is always 3), but this approach effectively lets us natively test predicates that accept, and should produce, multiple solutions in a very simple fashion. In programming contests there are situations where several solutions have to be handled by ad-hoc correctors or else the burden is passed to the contestant itself, by adding a redundant layer of complexity to the problem, asking the solutions to be presented in a specific order, or a solution with specific properties to be printed.

In concrete terms, we run a customized predicate in which the rule part is made of the conjunction of all the query calls we want to test and a final query which uses Prolog's I/O capabilities to write some simple sentence to the standard output (it can be for example a simple "ok"). So, this sentence is written only if all the queries are successfully satisfied, that is, if all tests are passed. If the expected output in Mooshak is exactly the sentence that we use, then the default evaluator gives "accepted" if and only if the program passes

all the tests, thus mimicking what happens in a conventional evaluation. If we want to make Mooshak aware of exactly which queries are passed, we can just create several input tests, each of them with the desired queries.

With this scheme, we can use Mooshak from the very beginning of the logic programming course. As with Haskell, students do not need to write complete programs in order for them to be tested by Mooshak. Actually, with this in mind, we have put together a set of more than 60 predicates and made them available through Mooshak permanently. Different subsets of this set are viewed as different "contests". For example, the "Lists" contest asks the students to program predicates to compute things such as the maximum of a list, the average of a list of numbers, the concatenation of two lists or the removal of duplicates. This invites students to become self-sufficient in their struggle with the language, relying on the automated feedback for validating the solutions they invent. We published additional problems that went beyond the goals of the course, as challenges to the most diligent students. Actually, some of these problems were taken from real programming competitions, namely the Portuguese Logic and Functional Programming Contest (CeNPLF, 2007). We observed that many students accepted the challenge and made an effort to solve those problems which, at the time, were on the limits of their programming capabilities.

Like in the case of Haskell, this competitive environment fostered a desire to solve all the problems proposed, in order to reach a higher position in the ranking and get the satisfaction of being up to the challenge. Indeed, having the program that you tentatively and hesitantly wrote to solve a difficult problem finally "accepted" by Mooshak creates a sense of fulfillment that we, as teachers, must not underestimate. In fact, students that would otherwise just look at the exercise and put it aside as too easy or too difficult, have now the incentive of actually programming it, gaining more experience and more wisdom in the process, because things are not usually as easy or as difficult as they seem. While we must contradict the pitfall of equating being accepted by Mooshak and correctness, we should note that the validation provided by the automatic judge does help many students to gain confidence in themselves and their emerging programming capabilities.

For the purpose of direct student assessment, in some years we have used Mooshak in "submission labs". Students were handed in a predicate, and they had to program it and submit it to Mooshak within a determined amount of time. If it was accepted, they would earn some points. This is very much like a programming competition, with an immediate, tangible result.

Mooshak was also used to guide students through more complex programming assignments. Instead of building a complete program from scratch on their own and testing it at the end, students follow directions to implement intermediate predicates, which they can test with Mooshak, before they advance. This way, students are more confident of the correctness of the building blocks of their programs. The intermediate predicates were graded directly by Mooshak, and the success of the overall assignment indirectly benefited from the automatic evaluation.

Typically, we leave all contests open, even after the course finishes. Students of the course can come back to the early exercises, in preparation for the exam, for example. We

also have had cases of students from previous years who resorted to the "old" and familiar Mooshak to refresh their Prolog, years later. This is made possible by an optional facility of automatically accepting new registrations, thereby opening Mooshak to all interested, when we are willing to allow that. Actually, some current students use that facility, which allows them to experiment their code anonymously.

5. Using Competitive Programming with Imperative Languages

The more conventional imperative programming languages can also benefit from schemes similar to those we describe for Haskell and Prolog. In fact, when we use Mooshak in courses for teaching programming in Java, C, C++ or Pascal, we usually do it in the ordinary form of having the submitted programs perform their own input and output. In other words, we prepare the contests in the normal way, setting them up for doing the automatic evaluation of the complete programs student will write. This is fine, but we cannot follow this approach in the very first labs of the semester, because students who are taking the first steps in programming do not have the knowledge to correctly construct a complete meaningful working program. For that, among other things, they would need to know how to use the input and output functions of the language, what is always tricky and confusing for beginners. Besides, they would have to respect precisely the specification of the output given in the problem description. We know by experience that this is difficult at first, when one is not used to that kind of painstaking detail.

Whereas with Haskell and Prolog we did not have input and output at the onset, with C, C++, Java and Pascal, we do, but we are willing to waive it. The beauty of this idea is that we can easily use and adapt Mooshak to evaluate only pieces of code and not complete programs, also for these more conventional languages. Technically it is even simpler than with Haskell and Prolog. We ask the student to program a certain function and we provide, inside Mooshak, a program that calls that function. For example: suppose we require a function that receives three real numbers as arguments and returns the average. The program we supply calls that function directly several times, with different arguments, each time writing the result on the output stream, exactly as we did with Haskell functions.

For doing this, we only have to tweak the compilation phase. Now we have not only the source file that was submitted, containing the definition of the required function, but also a complete program prepared beforehand with our own customized main function and all library declarations in place, and a header file with the prototype of the required function. Care must be taken in order to avoid such name clashes, but this approach is certainly feasible. Therefore, we compile the submitted source file plus the supplied program.

After this, running a program would simply be calling the executable created after the compilation. Since we wrote the main function ourselves, we can make it work the way we want. We can hardwire all the tests inside our own program, calling the evaluated function for each set of arguments to be tested, or we can write a loop that reads a set the arguments from a data file and calls the function with those arguments.

No matter what technique we choose, the remarkable fact to observe is that it is possible to do this in a very early stage: even the most primitive concepts and technique that we teach can be automatically evaluated, immediately. This can be done with any programming language, imperative, functional or logic.

This early introduction of automatic evaluation for imperative languages has a number of benefits, other than enticing students into the programming competitions, and the early exposure to the joys of having your program accepted by a robot, and earning points by it. First, it allows us to use the first classes to stress program development using functional decomposition, without the distraction of input and output. Second, it leads to more submissions being accepted in first labs, raising the spirits, and making students trust the teachers more that they would if what the teachers presented as an easy procedure proved to be embarrassingly difficult and distracting. In fact, the usual approach of submitting complete programs becomes very frustrating for beginners, precisely because very often the error is not in the function we want to test but in the part of the program that reads and writes. In this part, one has to worry about formatting details, about parsing the input line, about dealing to the end of data, and this can be overwhelming and is always distracting.

As a side-effect, this approach forces those students who have some knowledge of programming from secondary schools, and who are able to write long sequences of instructions, happily mixing input, computation and output, to drop that practice, and understand that the real challenges of programming are elsewhere.

6. Feedback from Students

We have used the approach described in Section 3 recently, in a first-year introductory programming course. There were students taking the course for the first time, but there were many repeating it who, having failed in the past editions, were taught differently. We initially observed frenzy amongst students, as they got acquainted with the automatic judge and began to become conscious of the way their assignments were to be submitted and graded. After a while, the agitation increased due to the perception that there seemed to be more assignments than usual. This was true: with the automatic judge, all the assignments were to be submitted and each successful submission did count, very little, in the final grade. Eventually, things calmed down, with practice and routine.

At the end we carried out a survey, to ascertain the students' reactions and overall impression of out approach. We had 115 responses, from more than 90% of the students taking the course. For most questions we used the 5-point agree-disagree scale. Table 1 summarizes the responses to the questions relevant to this paper.

In the questions, "platform" means the automatic judge, Mooshak, plus the learning management system that we were using, Moodle (Moodle site).

We observe that the students are not very assertive, except when giving advice to teachers, as in the forth question. For all questions, more than half of the students either "strongly agree" or "agree". Even for the third question, that shows a certain discomfort with Mooshak, only less than 20% of the students consider that Mooshak is not an effective learning tool.

Table 1
Results of survey, abridged

	Strongly agree	Agree	Neutral	Disagree	Strongly disagree	Don't know
Programming assigments are more interesting because they are supported by Mooshak	26%	50%	17%	6%	1%	1%
Assessment by the platform is generally fair	15%	62%	18%	4%	1%	1%
With Mooshak we learn more than without Mooshak	20%	35%	25%	11%	7%	2%
Other teachers should be encouraged to use a similar platform	61%	25%	7%	3%	2%	2%

This survey confirms results that we have observed in similar surveys for other courses, but is especially interesting because it was given to a class that used Mooshak for the first time, starting immediately on the first lab, with a functional programming language, as we explained in Section 3.

7. Conclusion

It is a cliché that computers excel at performing boring, repetitive tasks. For teachers, one of those boring, repetitive tasks is reviewing students' homework, especially when there are many students and many assignments. Computer help here is most welcome. In many disciplines, however, computers are not up to the task, at least not yet, but for programming, we are lucky that we can take advantage of tools like those used for managing programming competitions to take care of that chore.

Automatic judges have been designed mostly for imperative languages; they traditionally work by inspecting the input-output behavior of programs. Yet, we have been using them with Haskell and Prolog, which are not imperative, and for which input and output is not a technique that is learned in the earlier stages. For making automatic judges work with Haskell and Prolog we had to adapt them and in the process we discovered that, avoiding the complications of input and output, we were able to start using automatic evaluation much earlier in the courses. This possibility promotes a new approach of teaching, in which all the exercises and assignments, from day one are to be submitted to the judge, with immediate feedback to the students. This way, the productivity of students, measured in programs written and submitted, increases significantly. Also, the visibility of everybody's successes and failures aids in creating a challenging working environment and a healthy sense of community. This makes learning more enjoyable and more rewarding.

As a side-effect, students got acquainted with competitive programming, via the tools used to evaluate their programs. Many derive a special pleasure in appearing in the top places in the rankings kept by the automatic judge, even if that has no importance in the context of the course. We can expect that some of them will have been bitten by the competitive programming bug and will now create teams to participate in real programming tournaments.

References

Ala-Mutka, K. (2005). A survey of automated assessment approaches for programming assignments. *Computer Science Education*, **15**(2), 83–102.

CC2001 (2001). *Computing Curricula 2001, Computer Science Volume.*
`http://www.sigcse.org/cc2001/`

CeNPLF (2007). *Concurso/Encontro Nacional de Programação em Lógica e Funcional.*
`http://ceoc.mat.ua.pt/cenplf2007/`

Douce, C., Livingstone, D. and Orwell, J. (2005). Automatic test-based assessment of programming: A review. *Journal on Educational Resources in Computing* (*JERIC*), **5**(3).

ICPC site. *The ACM-ICPC International Collegiate Programming Contest.*
`http://icpc.baylor.edu/icpc/`

Ganesh site. *Ganesh – An Environment for Learning Computer Science*.
`http://www.dcc.fc.up.pt/~zp/ganesh/`

Leal, J.P. and Silva, F. (2003). Mooshak: a Web-based multi-site programming contest system. *Software Practice & Experience*, **33**(6), 567–581.

Moodle site. *Moodle – A Free, Open Source Course Management System for Online Learning.*
`http://moodle.org/`

Mooshak site. *Mooshak Contest Management System*.
`http://mooshak.dcc.fc.up.pt/`

SWERC (2007). *Southwestern Europe Regional ACM Programming Contest 2007.*
`http://ctp.di.fct.unl.pt/SWERC2007/`

Yap site. *Yap Prolog – Yet Another Prolog.*
`http://www.dcc.fc.up.pt/~vsc/Yap/`

P. Ribeiro is currently a PhD student at Universidade do Porto, where he completed his computer science degree with top marks. He has been involved in programming contests since a very young age. From 1995 to 1998 he represented Portugal at IOI-level and from 1999 to 2003 he represented his university at ACM-IPC national and international contests. During those years he also helped to create new programming contests in Portugal. He now belongs to the Scientific Committee of several contests, including the National Olympiad in Informatics, actively contributing new problems. He is also co-responsible for the training campus of the Portuguese IOI contestants and since 2005 he has been Deputy Leader for the Portuguese team. His research interests, besides contests, are data structures and algorithms, artificial intelligence and distributed computing.

P. Guerreiro is a full professor of Informatics at Universidade do Algarve. He has been teaching programming to successive generations of students, using various languages and paradigms for over 30 years. He has been involved with IOI since 1993. He was director of the Southwestern Europe Regional Contest, within ACM-ICPC, International Collegiate Programming Contest (2006, 2007), and chief judge of the worldwide IEEExtreme Programming Competition 2008. He is the author of three popular books on programming, in Portuguese. His research interests are programming, programming languages, software engineering and e-learning.