Building a Computer Program Grader

Don Colton, Leslie Fife, Randy Winters
School of Computing
Brigham Young University Hawaii
Laie, Hawaii 96762, USA
don@cs.byuh.edu

Abstract

Students often learn best by doing, and they may learn programming skills best by writing many programs, ranging from simple to complex. Overworked teachers can be dismayed by the prospect of grading still more programs per student as well as teaching introductory classes with ever larger enrollments. We present GradeBot, an automatic grader for computer programming lab assignments. The automatic grading approach offers substantial advantages and opportunities, but also some disadvantages and challenges. GradeBot evaluates student programs written in any of several languages, including C, C++, Java, Perl, Tcl, and MIPS assembler. Guidance for similar projects is provided through a discussion of the construction and operation of GradeBot.

Keywords: GradeBot, grading, programming, automated grading, testbed, C, C++, Java, Perl, Tcl, MIPS, SPIM, cheating

1.0 Introduction

In our experience, when intermediate-level programming students (in Computer Science or Information Systems) are given one programming assignment each week throughout the semester, they are generally successful at that pace of learning.

However, when novice programming students in a Programming I course were assigned at the same pace, the results were not good. By show of hands, 80 to 90 percent of each class claimed to have never programmed before in any language, and nearly all the rest had done only a few programs in Microsoft Visual BASIC. Many of the students experienced difficulty in completing the assigned labs. Because of this difficulty, some students gave up in frustration. Others in desperation acquired "extensive unauthorized help" which did not result in actual learning of the assigned material.

It was felt that inexperienced students were not successful with the pace of one program per week because it forced them to learn and demonstrate too much new material per program. Rather than giving even fewer assignments, it is felt that many more programs should be assigned, but with each demonstrating fewer new concepts. A change was made to better support the students by assigning and grading four or five programs per week instead of only one.

Although this seemed like the right thing to do for the students (and still seems so), it presented difficulties for the instructor. It created a huge grading burden for which GradeBot became the solution.

The thesis of the GradeBot project is that student learning in introductory programming classes can be effectively facilitated by the use of an automatic program grader.

2.0 Motivations

The initial and most important motivation was to support having students write more programs with a smaller increment in difficulty from each program assignment to the next. The desire was to do this without hiring more teachers or using more teacher time.

Starting from the idea of using a robotic* program grader, a few other expected benefits were identified: students would get faster responses to their program submissions, and distance-education courses might be taught at remote locations more easily. In addition students might engage in independent study or review with nearly no impact on instructor time.

(***Robotic:** It should be noted that the term "robotic" as used in this paper does not refer to the classical device that senses and manipulates its environment. Instead, robotic refers here to an automated process that was previously done by a human, and is still done in somewhat the same way that a human would do it. It is also an *homage* to the name "Robo Judge" that identifies the software used in some ACM Programming competitions.)

Paradigm Shift: It is important to note that automatic grading offers a complete paradigm shift from traditional grading. In traditional grading, the student turns in the assignment one time and a human grader evaluates it one time. In automatic grading the student is allowed to turn in the assignment many times without penalty, and the automatic grader evaluates each one quickly and patiently. Credit is granted when the student program actually works completely correctly, but not before that time. The student is not penalized for submitting ten or one hundred times before achieving a correct result. It is typical for each student to submit each lab about twenty times before getting it right. Most of the submissions are believed to incrementally resolve small problems such as output formatting, but this hypothesis has not been tested.

The opportunity to submit again and again is crucial to the operation of this system. Because there is no penalty for failure, students can be held to a higher standard, and required to try, try again until they achieve success. In such a setting, even small mistakes such as errors in spacing of the final printout, can be pointed out and not accepted. The students are told that in the real world, programs are required to perform with exactness and accuracy, and that testing against a suite of examples is common.

Ultimately this requirement of perfection seems fair because small and simple mistakes can be corrected in small and simple ways, while difficult mistakes can only be corrected with much careful thought. A human is not required to evaluate the seriousness of each mistake and to assign partial credit accordingly.

Tutors are provided nightly from 3:00 PM until midnight to help students who have difficulty.

While automatic grading offers a complete paradigm shift, it does not require it. It is still possible and desirable to involve a human grader as an evaluator of program style or to ensure that the program was written according to specifications. However, such involvement is not required.

More Programs per Student: Robotic grading would allow a move from 5 or 10 programs per student per semester toward a target of 50 programs per semester. Rather than the steep learning curve of one program for each topic, e.g., variables, if/else, loops, functions, and arrays, one might have many more programs, resulting in a more gradual learning curve. Each new program would introduce only one small concept rather than something larger.

More Students per Teacher: With robotic grading of most or all assignments and tests, it seemed that faculty could be more productive per contact hour by admitting more students into each class and teaching larger sections. The preparation time for a lecture promised to be about the same whether there were 15 students or 50 students.

Faster Response To Students: With a robotic grader in place, students would be able to submit their lab work and find out immediately whether it was "correct" or not. This seemed much better than collecting the programs in class on paper, or diskette, or sent by email, or deposited in a folder on the campus file server. Extensive hand grading was bad enough but managing and returning all the work with comments was also a burden.

Automatic Comments To Students: To the extent the robotic grader could evaluate

student work, it might also identify and coach in solving typical specific problems noticed for each student, such as forgotten newlines or extra whitespace (e.g., spaces, tabs, carriage returns, newlines), sometimes much more patiently and clearly than the instructor may have done.

No More Partial Credit: As mentioned above, one happy side effect of immediate response to students was the practical opportunity to require perfect programs from the students. Rather than guess how close they were to achieving the goal, they were simply told what test case led to their failure and told to fix it and resubmit. They were then left with the challenge of figuring out why their program behaved wrongly in that case. This seems more true to life.

Last-Mile Learning: By debugging their own programs, students engaged in "last mile learning." This is the learning that occurs when one finally finishes something, and does not merely imagine that it is "basically" finished. It is sometimes said that "the devil is in the details." By confronting that devil more true learning occurs.

Distance Education: It was imagined that the introductory programming course could be automated to such an extent that lectures could be recorded on video and the entire course could be delivered, conducted, and graded almost without human intervention. To make distance education possible, programming assignments were submitted by students through the Internet, originally by email. Because email is the most ubiquitous application on the Internet, this meant that the course could theoretically be conducted remotely to students anywhere so long as they had email.

Open Entry, Early Exit: It was believed that by using the Distance Education model, a tutor could handle questions and an instructor would be needed only rarely to resolve problems. Under this model, it would be possible to let students enroll at any time and complete at any time, and not just from start to end of semester. Assignment deadlines could be tailored to each student's personal timeline. This would allow particularly challenged students to take more than one semester to finish the class.

3.0 Grading Model

In this section, the grading model is considered. A progression of developments is presented here, showing how the grading engine developed to its current status.

GradeBot works by comparing the behavior of a student program to a defined standard. The behavior consists of the outputs that are produced by the student program. There is no attempt to "understand" the student program, such as would be done by a human that examined the source code. If the student program performs as required, it is declared to be correct, or "correct enough for our purposes today." If the student program fails, GradeBot can identify the discrepancy in the student program output but cannot identify the bug in the student program.

3.1 Source Code Submission

Students were to submit their programs as source code in any of the target languages. The following languages are currently supported: C, C++, Java, Perl, Tcl, and MIPS assembler (SPIM). Compilers (or equivalent) for these languages were available on the Linux system that was set aside to do the grading. As a first step, the program was compiled with all warnings enabled. If the program did not compile cleanly, it was rejected and the student was given no credit. The student was notified of this result.

Once the program was compiled cleanly, a series of zero or more tests would be performed. Each test followed a standard in, standard out evaluation model.

3.2 Standard In, Standard Out

The original grading concept was to provide two hand-made, hand-verified files for each test case. One would be the input (standard in) for the program. The other would be the desired output (standard out). The student program would be compiled and executed. The input file would be fed into the student program. The output results would be collected. Finally the collected results would be compared with the desired output. If they were identical, the next test would ensue.

3.3 Helpful Responses

In the ACM Robo Judge model mentioned above, contestants are only told whether they passed or not, and in case of failure whether there is a compiler error, a runtime error (such as divide by zero), a wrong answer, or a right answer incorrectly formatted.

For instructional purposes it was felt that if there were a discrepancy between the desired output and the actual output, the failed test case should be revealed to the student. This would allow the student to more easily debug his or her program. It also avoided most cases of students protesting that their program was actually right, they were absolutely sure. A counter-example served as very effective proof.

The UNIX **diff** command was used to compare the student program produced output to the desired "correct" output. The diff results were translated into plain English and reported to the student, saying: "Your first error is on line 5 of your output." GradeBot might add "Please check your spacing" or "Please check your punctuation" if it could identify that as the problem. Both the produced output and the correct output would then be printed so the student could compare.

Ideally the grader would point out the place where the student program was wrong, rather than the place where the output was wrong. Humans can often do this, but it is beyond the capabilities of this robotic system.

There was some discomfort that this was gradually revealing all the test cases to the students, and the students could then develop programs that treated each test case as a special case, hard-coding the output once the test case could be recognized. It seemed unlikely that students in the introductory classes would have this sophistication, but it was enough of an issue that it is addressed below.

3.4 Infinite Loops

Infinite loops were foreseen as a problem from the first. To deal with this, a timed execution facility called **timed-run** was used. It was already present on our Linux system, and is part of the **expect** package (Libes, 1995, p.17). Because the programs were simple and the processor was fast, it was felt that a few seconds should be enough clock time to do almost anything. Therefore, execution time was limited to two seconds in the general case. This has proven to be ample for all but a few special programs.

Not foreseen were infinite loops with print statements nested inside. The first occurrence was a program that generated 100,000 identical lines of output in the two seconds before it timed out. It took an hour to email the results back to the student, who was in just the next room.

Two measures were adopted to mitigate the infinite loop print problem. First, before mailing, identical lines were recognized and "compressed." Any time there were three or more lines that were identical, only the first would be returned, followed by a statement such as "the next 183245 lines are the same." This helped for the infinite identical print problem, but was not general enough.

The second measure was to look at the size of the desired output and use it as a guide for what was reasonable. It was decided that if the desired output consisted of n lines, the student would be allowed $2n+10$ lines and the rest would be counted and truncated. (The value $2n+10$ was chosen to allow students a reasonable number of extra lines of printout; n would be minimal, $2n$ allows some extraneous lines, and the $+10$ allowance handles the case of very short output files (small values of n). The important thing was to give the student a good view of his output without falling into an infinite output.) That was a more satisfying response. In four years there have been no further infinite emails, even though they are still possible if a student produces a single line that is infinitely long.

3.5 Program Crashes

Another problem was the core dump files that were created by student programs. Those were discovered to take up a substantial amount of disk space. To deal with them a nightly "**cron** job" was set up to remove all core files within the testing directory tree.

3.6 Machine Crashes

It was recognized that a clever and malevolent student could submit a program that would crash the GradeBot server. In C,

```
while (1) fork();
```

would be such an example. In our case such students can be identified and handled because GradeBot keeps a history of all submissions. If not, the input file could be prescreened to watch for specific constructs such as the word "fork."

One or two clever and motivated students have found ways to crash the server, but they have been proud of their achievements and have been willing to accept acknowledgement for their cleverness. They have not been an ongoing source of annoyance. In four years there has been no need to deal with this, and the plan is to deal with it when it becomes a problem.

3.7 Creating New Labs

To keep the programs from becoming too well known, with solutions too easily available, it seemed important that labs could be created and modified easily.

Initially the creation of new labs proved to be a lot of work, both for the detailed instructions that were prepared for the students and for the test cases that were prepared and verified by hand.

It was helpful to realize that with the test cases revealed to the students, there was little need for most of the file-detail instructions to the students, such as the exact format required in the output. The test cases were in effect the detailed instructions, at least to a level that might be acceptable for a first course. It was simple to augment the test cases with a paragraph or two outlining the task and report that to the student as part of the GradeBot response.

There are some tradeoffs with having detailed instructions. On the one hand, giving detailed instructions can help the student plan ahead and avoid frustration. On the other hand, many students do not understand the fine details in the instructions until they have a general version of the program working, and having a number of special cases delineated only serves to confuse them. On yet another hand, in many real-world scenarios, obscure test cases are only discovered in beta test or after product release.

By careful arrangement of the test cases, the fundamental program is typically tested before the special cases are revealed.

Although it was anticipated that new labs would be created frequently, in fact only a few new labs are created each year, mostly in response to new learning objectives rather than to avoid student cheating.

4.0 Grading Engine

With the original grading model, students could in *n* tries discover all *n* test cases being used, since there were a finite number, and *n* was generally small for hand-verified test data.

4.1 Plug-In Test Modules

To get beyond hand-verified test data and a relatively small number of test cases it was seen as more efficient, enjoyable, and reliable to code a program to test the student program. The program, called a plugin, would generate random inputs and matching outputs to test the student program.

As the plugin ran, each time it wanted input, the random number generator was called to create the appropriate input. The input was then saved for the student program and also processed by the plugin program. Each time the plugin generated output, it was saved for comparison against the student program.

For increased modularity, the plugin program was generally divided into two parts: a prototype program that behaves essentially like the student program, and a random input generator. This division of labor proved to be helpful.

In an actual test scenario, the prototype program would be run a number of times, usually ten to thirty times. After each run, the student program was executed and resulting outputs were compared. If the outputs matched exactly, the process continued. If not, the offending input/output

pair was reported back to the student and the process ended. If all the tests were passed, the student was sent a congratulatory message and the instructor was sent a completion message for entry into the grade book.

Appendix A includes specific details about the random input generation together with a complete and annotated example of a program to grade a simple student assignment.

4.2 Interactive Dialogue

Over time, the instructor was occasionally confronted by examples of student code that worked well enough for GradeBot but were still wrong.

One typical example of this would be a program to ask for a number, read it in, add one to it, and print the result. The student program could instead read in the number, add one to it, and THEN ask for the number and print the result. Using standard in and standard out destroyed the interleaving sequence, the "dialogue," between input and output. All inputs could be read first, and then all outputs created. But the intention of the instructor was to have inputs and outputs interleaved in a more reasonable fashion.

A major overhaul of GradeBot was conducted to get away from the batch input/output model. An interactive dialogue model was adopted for most program grading.

Instead of comparing a whole output file, the student program outputs were verified one line at a time, as they were generated. Similarly, the inputs were provided one line at a time as they were needed. With this improvement, the student could be forced to prompt for input before actually reading the input.

An unexpected benefit of this approach was the fact that infinite printing loops were no longer a problem. At the first sign of trouble, the student program was terminated and the remaining dialogue was modeled for the student. Only the first error line was reported.

4.3 Longer Outputs

With the advent of computer-generated test files, it because practical to have longer input and output files. When all inputs and outputs were hand-generated and hand-verified, there was a strong tendency to keep things short and simple. This resulted in toy tests.

At the same time, some programs could take much longer than the two seconds traditionally allowed. For example, one lab requires the students to download a web page, parse it for links, and then download the files it is linked to, eventually building a complete site map of a web site. This lab could easily run five minutes for a web site with one to two hundred web pages. For another example, given a set of classes to schedule, together with the semesters in which each class is offered and the prerequisites for each class, the student was required to find a shortest possible graduation plan. For most inputs the program was very fast, but for some inputs the processing was NP complete.

GradeBot was easily modified to allow longer timeframes. The prototype program was timed and the student program was allowed $2t+10$ seconds to run, where t was the time taken by the prototype program.

4.4 Throttle

GradeBot was built on a submit/reply model. Students came to expect the reply within a second or two. Occasionally there would be a program which legitimately took longer than a few seconds to run. In such a case, the student was supposed to wait until the response came back.

Of course, students are about as patient as most people. This means that when the answer did not appear after five seconds, they would assume the program did not submit properly, and would submit it again. And again. And again.

It was discovered that a single student could submit a long-running lab perhaps dozens of times, and GradeBot would dutifully try to run them all simultaneously. As the server did its context switching from one task to another, thrashing would result. This would make the response time even slower for

everyone and eventually it led to very long delays. Finally, the original student would get back several replies over a span of several minutes. Most of the replies were redundant.

This tended to happen a lot toward the end of the semester, when the most complicated programs were due, and as students were frantically trying to complete as many projects as possible before the deadline.

To solve the problem, GradeBot creates a "lock" (implemented as a zero-length file in a special directory) while a student program is processed. If a subsequent request is received from the same student, it also creates a lock. As long as the new lock is not the oldest lock, GradeBot sleeps a few seconds and checks again. Finally, the new test runs and the lock is deleted. Additionally, if GradeBot decides to sleep, it sends an email back to the student stating that GradeBot is still testing a lab that the student previously submitted, and as a matter of policy the labs will be done one at a time.

To further prevent multiple submissions of the same program, for long-running tests GradeBot would send an early reply stating that the first few tests were successful and the longer tests were starting, and please do not submit again for at least 20 minutes, or until you get the rest of the results.

This resolved most of the difficulty from duplicate requests. However, the multiple email responses complicated matters for the web interface that was eventually built.

5.0 Worrying About Cheating

Some students were able to complete the labs but were still unable to perform on programming quizzes and tests given in class. Interviews with the department-provided tutors revealed the unsurprising fact that students were helping each other. Such help was explicitly forbidden.

At first, the instructor response was frustration and indignation, but this did not solve the problem. Entire classes were berated as a group to eliminate this cheating. It may have felt good to the instructor but it did not help to solve the problem.

There seemed to be two distinct elements contributing to the forbidden behavior. First, students seemed less upset about cheating in their interactions with a machine than they would in their interactions with a fellow human. Computer games often have "cheat codes" that can be downloaded. To many students, using cheat codes is acceptable.

Second, as demonstrated by the 2001 GRE CS Subject Test cheating scandal, in some cultures there is a strong us-versus-them mentality relating students to teachers. Students are culturally expected to assist each other, even when in defiance of instructor mandates. This cultural issue was more difficult to work around, and eventually the best solution seemed to be the formal acceptance of group work as a valid way to study.

5.1 An Age of Miracles

To identify cheaters, GradeBot incorporated a complete history of all lab work ever submitted by students. Each submission is converted into a standard form by, for example, compressing whitespace and removing string constants. A checksum is taken of the resulting code. When a student program completely passes a test, this checksum is stored in a database. When a new student program is submitted, this checksum is compared with the database. If a match is found, the full programs are compared. If a match is still found, an incident report is emailed to the instructor. The incident report details the "miraculous" fact that two programs were identical.

The initial result was lots of email. It was concluded that for a fairly simple lab, or for a lab that represented only a small change from sample code given in the textbook, the odds of duplicate programs were quite high. This was also true for programs that were explained thoroughly in class by the instructor or in the lab by the tutors. Not all of this activity could be called cheating.

The next step was to look at the predecessors to any code match. For each match, the miracle report was modified to list all the previous identical submissions that had been received. If many students shared the same code, generally there was a structural reason for that. If only one or two students

shared the same code, it was much more defensible to say that the students must have gotten it from each other. Still, one incident was enough to be cautious, but did not provide enough evidence to "convict."

The next step was to modify the miracle report to include past incidents of identical code involving that student. This turned out to be **very** helpful. When student A had code that was miraculously like that of student B on one assignment, and like that of student C on another assignment, and like that of student D on yet another assignment, it could be attributed to the fact that there were a limited number of common ways to write the program, given that the students attended the same lectures and visited the same tutors. But if student A had code like that of student B on quite a few labs, this indicated a fairly strong level of collusion.

5.2 Per Student Customization

Before the decision to lighten up on the apparent cheating problem, GradeBot was modified to allow each student to receive a similar but not identical problem when compared to his neighbors. The goal was to provide better evidence against cheaters because they could not use the excuse that they were solving the same problem. If identical submissions were detected, the source of the original program could be more easily and reliably identified and a punishment could be more fully justified.

This provided an interesting diversion during the development of GradeBot, but did not meet its goal of better identifying and punishing offenders. Recently developed test programs generally take no advantage of this feature.

5.3 Overcoming Cheating

The ultimate result of all the worrying about cheating was a conclusion that technical means could detect simple forms of copying, but effective police action could not be maintained because of the cultural desire to work together and the ease with which students could modify their copied work just enough to avoid being caught. For these reasons it became easier to quit trying to directly controlling cheating on the labs and to instead rely on testing in a controlled setting. A large share of the final grade now rests on in-class tests. Students are explicitly permitted to do their lab work in concert with anyone they want, but are reminded that one important goal is the learning they will need to demonstrate on the in-class tests.

6.0 Results

GradeBot has been operational for four years, handling an average of 400 students per year, each submitting roughly 1000 lab assignments to complete 50 labs per class, mostly in the Programming I and Programming II courses. It has been used with a variety of student programming languages, including C, C++, Java, Perl, and MIPS (in the computer organization / architecture class).

Instructors are very pleased with this tool, and desire to see it continued, but they are not totally satisfied. There are tradeoffs. Because faculty are not required to see every submission by every student, they tend to lose touch with the abilities of their students. Additional tools not reported here have been implemented to allow the teachers to monitor the progress of their students and identify those that are falling behind. Also, because student work is not reviewed by another set of eyes, there are stylistic issues that are not well addressed, such as commenting and indenting. Additionally, students can sometimes short-circuit an assignment, but writing a single routine to achieve a goal when the assignment was to create and use certain subroutines or data structures, or do something else in a particular way.

Students have reported having a love/hate relationship to GradeBot. Most students love the fact that they get immediate feedback, and can know that their assignment is completed and accepted for full credit. A few students hate the fact that GradeBot requires extreme attention to such details as spelling and spacing in their output, and that occasionally the appearance of blank lines in the output can be hard to plan (e.g., should the blank line print outside the top of the loop, inside the top of the loop, inside the bottom of the loop, or outside the bottom of the loop).

The quality of student programming skills seems to have improved a lot, but there is no control group, so this improvement must be regarded as anecdotal evidence. The fact that the students who complete the introductory classes have generally become capable programmers supports the hypothesis that automatic grading is a feasible approach to working with introductory programming classes.

7.0 Future Work

The GradeBot core provides evaluation of one assignment for one student at a time. Beyond this basic ability, a web interface has been created for students, and is very popular and is being used with substantial success. Also, a rudimentary instructor interface has been created and is being used. These tools have made GradeBot more convenient. Early versions required the instructor to be a programmer / hacker, and the current version still requires such a person to provide maintenance between semesters and to solve special situations that arise. The local software engineering class is planning to develop a better instructor's interface. Both the student web interface and the planned instructor interface may be reported in the future.

Additionally, with proper packaging this tool might be released to a broader audience, and it may become feasible to conduct a controlled study to see what quantitative effect this learning method has in comparison to hand-graded programming assignments. Interested parties are invited to contact the authors.

References

Chang, Carl, Peter J. Denning, James H. Cross II, Gerald Engel, Robert Sloan, Doris Carver, Richard Eckhouse, Willis King, Francis Lau, Susan Mengel, Pradip Srimani, Eric Roberts, Russell Shackelford, Richard Austing, C. Fay Cover, Gordon Davies, Andrew McGettrick, G. Michael Schneider, Ursula Wolz (2001). "Computing Curricula 2001 Computer Science," Final Report, 15 Dec 2001. Jointly published by IEEE-CS and ACM.

Libes, Don (1995). Exploring Expect. O'Reilly. ISBN: 1-56592-090-2.

Molluzzo, John (1996). C for Business Programming. Prentice-Hall. ISBN: 0-13-482282-X.

Ousterhout, John (1994). Tcl and the Tk Toolkit. Addison-Wesley. ISBN: 0-201-63337-X.

Appendix A

This appendix provides details of the random input generator together with a simple annotated example of a program grader plugin.

Tcl/Expect was chosen as the language for GradeBot for four reasons. First, a string-oriented language was desired because the grading process would require the generation and comparison of strings. Second, because of the large number of programs to be graded, it was felt that each grader should be stored in some sort of library and plugged in at run time. Third, the primary author of GradeBot had recently done a lot of programming in Tcl and wanted to build on this recent experience. Fourth, (after Tcl was selected) it was discovered that **expect** was a superset of **Tcl** and provided a good way to communicate with the student program as a separate process.

Perl would probably be another good language for implementing such a project.

A.1 Random Numbers

Tcl does not provide a native random number generation facility. The following procedure was developed to provide this capability.

```
# returns a 15-bit integer: 0..32767
proc random15 {} { global _R
set _R [expr $_R * 1103515245 + 12345]
expr int ( $_R / 65536 ) % 32768 }
```

The plugin for grading a lab provides a correctly functioning prototype of the student's program. As the prototype program runs, each time it wants input, the random number generator is called to create the appropriate input. The input is then saved for the student program and also processed by the prototype program itself. Each time the prototype generates output, it is saved for comparison against the student program output.

The prototype program was then run a number of times, usually ten to thirty times. After each run, the student program was compared. If the outputs matched exactly, the process continued. If not, the offending input/output pair was reported back to the student and the process ended. If all the tests were passed, the student was sent a congratulatory message and the instructor was sent a completion message for entry into the grade book.

The following procedures were developed and found useful for the creation of random inputs:

random(low,high) returns an integer uniformly distributed between low and high.

pick(list) returns a random element of the list. Each element is equally likely to be returned.

permute(list) returns a uniformly random permutation of the list.

rlog(low,high) returns a number between low and high, uniformly distributed in the log domain, that is, equally likely to be between 10 and 100 as between 100 and 1000.

random15 returns a 15-bit uniformly random integer (0..32767).

As a measure of relative usefulness, out of 85 test programs in use last semester, random is used 222 times, pick is used 132 times, permute is used 33 times, rlog is used 28 times, and random15 is used (directly) 3 times.

A.2 Sample Plugin

Following is an annotated example of a lab assignment test program. This program is based on a programming problem (chapter 1, problem 6) in Molluzzo (1996, p.22). Lines have been shortened to fit this paper.

```
proc sim in { global lab; start
  get "Type in four letters: "
  put $in
  set c3 [string index $in 2]
  get "The third letter was $c3.\n"
  runLog $lab [list sio [eof]] {st 0}
}

proc lab$lab {} { global lab errCt
  sim "wxyz\n"; # free sample
  if { $errCt } return; sim "abcd\n"
  do 5 {
    set ab "[pick a b c][pick d e f]"
    set cd "[pick g h i][pick j k l]"
    if { $errCt } return; sim "$ab$cd\n" }
}
```

The entire sample shown above is stored as a file in the **lab** directory for the **cs101** course. The file is **sourced** (read) into GradeBot when the lab assignment has been identified.

The file contains two procedures, **sim** and **lab$lab**. The **sim** procedure is intended to perform (simulate) one complete test of the student program. The **lab** procedure calls **sim** some number of times to perform a variety of individual tests of the student program.

With rare exceptions, test programs are written in Tcl, the "tool command language" invented and developed by John K. Ousterhout (1994) that forms the basis for the **expect** utility mentioned above.

proc introduces a new procedure. **sim** is the name of the first procedure. **in** is the sole formal parameter to that procedure, and is passed by value. Curly braces enclose the body of the procedure. **global** introduces a global variable, **lab**. All other variables are local. **start** calls another procedure to prepare the input and output capture routines.

Customary usage is to provide a **sim** routine for each test program, and for its parameters to be the varying elements of a test case. In this example, the sole element of the test case is a character string that will be presented to the student program as input.

get specifies a string (in this case a prompt) that must be presented by the student program. In this case, the prompt is "Type in four letters:" followed by a space but no newline.

put specifies a string that will be given as input to the student program. **$in** is the formal parameter, used as the source of information.

set is the assignment operator in Tcl. **c3** is the variable name (expressed as an Lvalue, a name to which a value can be assigned, typically occurring on the "L"eft side of an assignment statement). The square brackets enclose another command that will be executed, and whose results will be taken to initialize the variable **c3**. **string index** is a built-in command that will, in this case, extract character number 2 (counting from zero) in the string stored in the **in** variable.

get again specifies a string to be gotten from the student program. Slash n (\n) indicates a newline (carriage return, or line feed, or both).

The previous commands have prepared the input-output script to be carried out. **runLog** uses **expect** to carry out the script as a dialog with the student program. Each student output is compared to the expected value. Each time the student program should be waiting for input, the script provides the input to be given.

The second procedure, **lab$lab**, has a global variable **errCt**. So long as this counter is zero, testing continues. If **errCt** becomes non-zero, testing will end and credit will be denied. **sim "wxyz\n"** provides the free sample of input and output the student will be shown to help them understand the task. It is provided even if the error count is non-zero (for instance, if the compile failed).

The next simulation is provided only if the error count is still zero. The second set of input will be **"abcd\n"**.

do 5 is a shortcut procedure unique to GradeBot that means "perform this loop five times."

pick a b c will return one of those three letters, each with a probability of 1/3. There are 81 possible strings that can be generated in this loop. So long as **errCt** remains zero, additional strings will be tried, up to a limit of five (do 5).

When the end of the **lab$lab** procedure is reached with **errCt** still equal to zero, the student will receive credit for completing the lab.