Building a Computer Program Grader

Don Colton, Leslie Fife, Randy Winters, Jim Nilson, Kurt Booth
School of Computing
Brigham Young University Hawaii
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 consider GradeBot, an automatic grader for computer programming lab assignments. Such an 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, automation, testbed, C, C++, Java, Perl, Tcl, MIPS, SPIM

1 Introduction

In our experience, when sophomore-level Computer Science students are given one programming assignment each week throughout the semester, they are generally successful at that pace of learning.

However, when inexperienced freshman-level students from Computer Science and Information Systems 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.) 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 is believed that inexperienced students are not successful with the pace of one program per week because it forces 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, each demonstrating fewer new concepts. Locally a move was made to better support the students by assigning and grading 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 is that student learning in introductory programming classes can be effectively facilitated by the use of an automatic program grader.

2 Motivations

The initial and most important motivation was to support having students write more programs with a smaller increment in difficulty from each 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.

More Programs Per Student: Robotic grading would allow a move from 5 or 10 programs per student per semester toward a target of 75 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 could 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, sometimes much more patiently and clearly than the instructor may have done.

No More Partial Credit: A 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. They were then left with the challenge of figuring out why their program behaved wrongly in that case.

Last-Mile Learning: By debugging their own programs, students engaged in "last mile learning," which is the learning that occurs when you finally finish something, and do not merely imagine that you have basically finished it. It is sometimes said that "the devil is in the details." By confronting that devil 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 this 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 Grading Engine

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

3.1 Standard In, Standard Out

The original grading concept was to provide two 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. If the compile failed, the student would be notified. Otherwise 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. If there was a discrepency, the failed test case could be revealed to the student so that a fix could be prepared.

3.2 Helpful Responses

Rather than simply telling the student that their program had failed, it was decided to reveal the actual test case that brought about that failure. Students who particate in ACM programming contests are very often **not** given the final test cases, but it was felt this would be too difficult for students in Programming I and II.

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 them 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.

There was some discomfort that this was gradually giving away all the test cases to the students, and the students could 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 annoying thought that it was addressed below with Random Test Cases.

3.3 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 occurrance was a program that generated 100 thousand identical lines of output in the two seconds before it timed out. It took an hour to email the results back to the student. It was very amusing, but a correction was sought immediately.

Two measures were adopted to cure 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. 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.

3.4 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 and to deal with them a nightly "`cron` job" was set up to remove all core files within the testing directory tree. This continued to be an important tool until the advent of Interactive Dialog mentioned below, and is still in use since it is more trouble to verify that it was safe to dismantle it than to let it run.

3.5 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 pre-screened to watch for specific constructs such as the word "fork." The bottom line is that 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.

3.6 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 this 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.

Two realizations were helpful. First, there was no need for most of the instructions to the students since the test cases were in effect instructions, at least to a level we felt acceptable for a first course. It was simple to augment the test cases with a paragraph or two outlining the task and report that back to the student as part of the GradeBot response.

The second conclusion came with the Random Test Cases issue.

3.7 Random Test Cases

Because students could in n tries discover all n test cases being used (if there were a finite number), and n was generally small for hand-verified test data, it was seen as more efficient, enjoyable, and reliable to code a prototype program to perform the target task. This program was then paired with a task-specific input data generator. In rare cases the two were merged into a single program.

Tcl/Expect, the language in which the bulk of GradeBot is written, 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 }
```

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

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 gradebook.

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

`random(low,high)` to return an integer uniformly distributed between low and high.

`pick(list)` to return a random element of the list.

`permute(list)` to return a random permutation of the list.

`rlog(low,high)` to return 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` to return a 15-bit 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.

3.8 Sample Test Program

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 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). 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 indicates a newline (carriage return).

The previous commands have prepared the input-output script to be carried out. `runLog` carries them out and reports the results.

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.

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.

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.

3.9 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.

At first, the instructor response was frustration, upset, and indignation, but this did not solve the problem. Entire classes were berated as a group to eliminate this cheating. It did not work.

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 interacting with a fellow human. Computer games often have "cheat codes" that can be downloaded. To them it is no big deal.

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.

3.10 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 detailed 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 even true for programs that were explained thoroughly in class by the instructor or in the lab by the tuturs. 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 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 got it from each other. Still, one incident was enough to be cautious, but not enough evidence to be pursuasive.

The next step was to modify the miracle report to include past incidents of identical code. This turned out to be very helpful. When student A had code that was miraculously like student B on one assignment, and C on another assignment, and 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 the students attended the same lectures and visited the same tutors. But if student A had code like student B on quite a few labs, this indicated a fairly strong level of collusion.

3.11 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 neigh-

bors. 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 inputs occurred, the source could be identified and a punishment could be 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.

3.12 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 give up on the labs and to rely on testing in a controlled setting. A 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.

3.13 Interactive Dialogue

Over time, the instructor was occasionally shows examples of code that worked well enough for GradeBot but was still wrong. The most notable 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 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. The new model was interactive dialogue.

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. Finally the student could be forced to prompt before 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 one line of errors was reported.

3.14 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.

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. This would make the response time even slower for everyone, and eventually 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 would happen a lot toward the end of the semester, 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) when a student program starts being 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.

This resolved most of the difficulty from duplicate requests.

4 Grade Management

Part of the model for GradeBot was a video game model. In this model, players complete a "level" and then move on the beat the next level. The intention was that as each program was completed, the student would be emailed some encouragement along with the challenge to complete the next lab.

This encouragement was provided in the form of a Status report listing all the labs completed and pending, and giving the student a projected grade for the course (at the current rate of achievement).

Initially students were guaranteed a grade of A once they reached 930 points (93%). Students would even try to complete labs after they took the final exam, just to get their total points up to the next level.

The status report was very successful and encouraging the fast students. It was also very successful at alerting the slow students to the fact that they were falling behind.

4.1 Variable Due Dates

As mentioned above, due dates can be customized per student. This was intended to allow each student to have his own timeline for completing the course. In practice it has been used to relax the timeline for a student who became ill or was traveling as part of a sport team. It is a little-used feature.

4.2 Counting Attempts

Initially it was thought that students should be graded partially by the number of attempts that were taken before the program was correct. Accordingly, GradeBot counts each attempt. However, the students expressed a great deal of gratitude that they were allowed unlimited submits. The best reason for limiting submits is to save the time of the grader, but this reason is not relevant in the GradeBot world. Another reason is to encourage students to do their own testing. For most of the students, we were delighted that they would get the work done and did not feel any strong need to threaten them by taking off points for each extra submission. It is a little-used feature.

Some high-achiever students do take a certain pride in solving a problem within some small number of submits. We do report on each submit how many previous submits have been processed for this student.

4.3 Testing Center Integration

We have the good fortune to have a university-sponsored testing center. Here we can deposit bubble-sheet tests for student to take at their convenience. We can also attach through telnet and download the student scores. This has proven to be very beneficial, as we were able to include testing center activities as part of the status report.

4.4 Student Status Reports

The student status report for any given student lists all the assignments for which they can earn credit. For each assignment, the assignment name is given, together with the points earned or still available, and the date on which the points were earned or on which the assignment is due. A brief description of the assignment is also given. For tests, the test score is also shown. The status report is presented in a tabular form with one row per task.

In addition to the details for assignments, there is a header and a footer. The footer summa-

rizes the dates for that student, including the deadline for the final exam and the last day on which work can be submitted.

The general approach to late work and due dates is that each assignment should have a due date, but it should be soft in the sense that when a student misses that date, they can still turn in the work a day later for a small penalty in points earned. The approach currently used is to take off one point per day late, up to a maximum of 40 percent off. Late work is always worth at least 60 percent of full credit, until the end of the semester.

At the top of the status report, the course grade for the student is forecast based on the student's performance on assignments completed and assignments due yet uncompleted. The projected score for unfinished assignments is based on the percentage performance for completed assignments of that type, together with a penalty for projected late work or work estimated to be completed too late to be turned in for credit.

Projected scores for all students are kept in a separate file, and each time a status report is prepared, the score is compared to that file to determine the rank of that student. For students in the top half of the class, the status report includes their rank, such as "your rank is 3 out of 21 students." For top students, this is another motivator as they seek to be number one in the class. For students below the top half, this was believed to be a non-motivator and was therefore not presented.

4.5 Teacher Status Reports

The teacher of each class receives several daily status reports summarizing the performance of the students in his class. One report shows the labs completed in grid form, with students listed down the side alphabetically and labs listed across the top chronologically. At a glance the teacher can tell which students are working ahead, and which students are selectively skipping labs.

Another report shows the activity level for each student across the past three weeks, on a grid that is filled in with the number of submits done per day, if any. At a glance the teacher can tell who has stopped working.

5 Web Interface

To make a faster start each semester, a web interface (sometimes called WebBot) was designed and implemented for GradeBot. The web interface has become so popular that students do not discriminate between the interface and the grading engine. It is all the same to them.

5.1 Before The Web

Before the web interface, the first week or two of class was spent getting students to log in and learn how to use a text editor, typically emacs. Then they learned how to email their submissions to GradeBot and how to receive the responses. This was frustrating to the teacher because it took so long before students could start to experience the excitement of getting work done and seeing their projected grade.

5.2 A Faster Start

The web interface was envisioned as a faster start. Students would be able to submit programs on the first or second day using familiar html forms with textareas and drop-down menus. The web interface was an incredible success.

5.3 A Rich Menu

to be added

6 Routine Maintenance

to be added

6.1 Per Semester

to be added

6.2 Per New Lab

to be added

6.3 Per New Server

to be added

6.4

GradeBot represents a case study of such facilitation efforts. It is set forward, together with observations about its effectiveness, the opportunities it creates, and the difficulties that occur. The project is deemed a success, and is believed to improve learning for most students, while using faculty resources more efficiently, but some difficulties remain.

A computer program grader is itself a computer program designed to evaluate student work in computer programming classes. Use of such a grader can allow the faculty member to be both more efficient and more effective.

For us, the typical first class enrollment is about 200 students per year, but upper division courses typically have enrollments closer to 20 students per year. Nearly all the attrition was between the first and second class.

My goal was to make the introductory class more of a "pump" and less of a "filter." A pump is a class the attracts students into the major and pushes them forward toward completion. A filter is a class that weeds out the students who are unlikely to be successful within the next few years. I felt that the first class was too early a time to be filtering students in all but the most obvious cases.

To feel successful, a student must earn a good grade and feel that the grade is legitimate. The student must feel mastery of the course material. For a few students this mastery comes almost without effort, as though by gift. For most of our students, this mastery comes after a mightly struggle.

To aid their success, we have staged this struggle as the conquest of a number of small programming assignments. We have tried to take small steps because the larger steps become stumbling blocks for most students. But smaller steps mean that there must be more of them. We do not feel comfortable having one program for each topic, e.g., variables, if/else, loops, functions, and arrays. The best students would be fine with such an approach, but we are trying to reach the rest of the students.

(It may prove that this effort is misguided, and the students we retain simply drop out later, thus wasting more of their own time as well as slowing down the natural achievers.)

7 Instructor Experiences

GradeBot was used by its designer in his courses, but over time other teachers were coerced into using it because they taught the introduction to programming class. It is typical that the designer and developer of a tool find it convenient and easy to use while others may not enjoy the same experiences. This section explores some of their experiences.

...

8 Thesis

9 Clueless

"I hear and I forget, I see and I remember, I do and I understand." — Confucius

I teach programming. I probably started out by explaining to students. This is how to write a program. This is how to design useful variable names. See the glorious symmetry of "if" and "while." Appreciate the wonders of call by value and call by reference. That was the way that I remembered my own introduction to comput-

ing. When I tried it in the classroom, the students all watched with eager eyes and smiled in appreciation. Then came the first test. Most of them had no clue what I was talking about.

Brigham Young University Hawaii has a largely international student body, drawing many students from Asia and Polynesia and less than half from the US mainland. Based on a show of hands at the start of each semester, eighty percent of students entering either the Computer Science major or the Information Systems major here report that they have never programmed before. Fifteen percent report one previous programming class (usually in Microsoft Visual BASIC) but assert that they really do not remember much. Five percent have more substantial experience.

There are many approaches to teaching computing. "Computing Curricula 2001 Computer Science" (Chang, et al, 2001) mentions in its overview of curricular models (chapter 6) the following introductory course approaches: Imperative first, Objects first, Functional first, Breadth first, Algorithms first, and Hardware first. In section 7.2 it states: "One of the most hotly debated questions in computer science education is the role of programming in the introductory curriculum."

We have adopted the "Programming first" approach for our CS and IS majors based on the belief that most of our students will not really understand anything until they have some programming experience. Our approach is to get students programming early in the hope that later we can address principles of software engineering and correct any habits that need improvement.

Large projects tended to overwhelm my Programming I students. Things may look easy when the instructor demonstrates, but when the students try it later, they find themselves lost. I decided to focus on teaching programming by giving lots of small programming assignments.

After collecting the first assignment and grading it, several things were clear to me. (1) Grading takes a lot of time. (2) I was using the same test cases over and over again. (3) Most of my time was spent reading broken programs and assigning partial credit based on how close the student seemed to be to solving the problem.

I automated my testing by creating input and output files. Each student program would be fed each input file. Output would be collected and compared to the standard. Correct outputs would match. Incorrect outputs would not match. Redirected input and output were cobbled together into a script (batch file).

```
c:\> stuprog < input.1 > output.1
c:\> stuprog < input.2 > output.2
c:\> stuprog < input.3 > output.3
c:\> diff output.1 correct.1 >> err
c:\> diff output.2 correct.2 >> err
c:\> diff output.3 correct.3 >> err
```

This approach worked well for correct programs. I could quickly determine that they were correct enough to satisfy me. But this approach did not work well for incorrect programs. I was still forced by my teaching ethics to examine each student's program and assign some amount of partial credit.

At some point I came to several conclusions. (1) Substantial learning occurs in the "last mile" of programming and debugging toward making a program work. (2) Students whose programs almost worked did not get this "last mile" learning. (3) I could save a lot of my personal time by automating the grading process such that the students could check their own work.

My idea was to prepare a grading testbed and let each student test his own program. Students should see their errors immediately so they could continue debugging. Students should see their errors without my intervention, so my time would be free for other needs.

10 Disadvantages in Advance

> Two roads diverged in a yellow wood,
> And sorry I could not travel both
> And be one traveler, long I stood
> And looked down one as far as I could
> To where it bent in the undergrowth;
> – Robert Frost

Before ever building the project, I was aware of several disadvantages. In my zeal I felt those were far outweighed by the advantages I hoped to gain. In retrospect I am no longer sure. A properly designed experiment would use a control group and measure the difference in performance, but some things are just not practical.

10.1 Graphics

My approach handled input and output in the traditional channels. It did not comfortably support graphics. Some of my colleagues are very excited by the use of graphics, particularly in languages like Visual BASIC and Java, to stimulate student interest. Male students in particular may be motivated by a desire to create video games. These generally involve a lot of graphics. Grading is also made easier for graphical programs because the expected behavior of the program can be readily observed.

I still have no idea how to automate the grading of graphical outputs in any useful way.

Even though GradeBot does not evaluate graphical output, there is nothing to require that all labs use GradeBot. Indeed tests are already outside the robotic grading system and graphical programs could be graded outside the robotic system as well.

10.2 Interaction

My initial approach provided all inputs through STDIN, as though they were typed from the keyboard while wearing a blindfold. Similarly, all outputs were captured as though they were printed and then taken away to be compared. This works adequately for many programming tasks, but I found it frustrating when there was to be some sort of dialog between the student's program and its user. This problem was eventually solved.

10.3 Developing Test Cases

It is important for students to develop their own test cases for the programs they write. GradeBot subverts this objective by providing all the test cases needed to earn credit.

To some extent, this is good for new students. A student writes a program. GradeBot finds a test case that breaks the program. The student finds and corrects the bug. GradeBot finds another problem. The process spirals outward from the most straightforward test cases until we reach the more obscure test cases. Students develop an appreciation for the fact that there could still be one more bug lurking in some obscure situation that they have not yet considered.

In the case of advanced students, GradeBot can be seen to simply perform a set of black-box tests. The student should be the one who is constructing white-box test cases. Test cases should use knowledge of the boundary conditions that exist in the student code.

Canned test cases are thus a mixed blessing.

11 GradeBot Version 1.0

Accordingly, I undertook to automate the process. At first I was spending days automating a task that formerly took hours, but it was enjoyable work and I persisted.

The first thing that I automated was the submit/reply process. Because it made sense to me, I organized it so students would send an email message to a special address. The subject line would identify the program to be graded. The body of the message would be the source code.

Upon receipt, the program would be extracted, compiled, and tested against the canned inputs and outputs. The results (output differences) would be emailed back to the sender (the student) and to me. If the results were perfect, the student would receive credit for the work. If not, the student was required to correct their program and resubmit until it became perfect.

This solved the "last mile" learning problem for me. It also got rid of partial credit grading. And importantly it allowed the students to submit their work at any time, day or night, and get immediate feedback.

It also allowed me to simplify my assignments. Rather than describing the output in excruciating detail, I simply assigned to students to "make it work." Variations in spacing were caught by my robotic grader and returned to the student. Students complained because the spacing had to be exactly right, but I countered that when you are printing checks (for instance) or anything on preprinted forms, spacing had to be perfect. I also pointed out that the end user often wants to control the exact appearance of the output but does not care about how the answer is generated. My robot was acting like a typical user, but was much more patient.

The students continued to have a love-hate relationship with the robot. They loved the fact it was available 24 hours a day, seven days a week. They loved the fact that they could fix their mistakes and try again for full credit. They hated the fact that "close" was no longer good enough, and they had to achieve that "last mile" learning. I continued to smile with delight that things were working out so well.

12 Revealing Answers

Part of the brilliance of my scheme was the fact that I did not have to explain in detail why their code was wrong. I could simply point out a sample input and output that failed and say, "your program did this, but it should have done that instead." I did not have to say "you should have used a loop here," or "you are clobbering your variable over there." The early version of GradeBot sent back (1) the original program, (2) the input that was given, (3) the output that was expected, and (4) the output that was actually received.

I felt it was necessary to tell them where their program failed by giving an example of that failure. This created a challenge for my scheme because students could program the answers instead of the process. If input is x, print answer y. No calculation was necessary.

I moved to the next step: random inputs. Instead of having a set of canned inputs and outputs, I decided it was important to generate random inputs (within certain ranges). I accomplished this by writing a "correct" version of the program in order to turn random inputs into the correct outputs. I also had to customize the random inputs for each problem so that the test cases were reasonable.

By doing this I escaped from the problem of memorized answers. I could create practically an infinite number of test cases, which was far more than the student could program by "special case" logic. Students were forced to actually write the programs assigned.

13 Cheating

Student programs were not reviewed by human eyes in every case. As time went on, programs were examined less and less frequently. I started to suspect problems with plagiarism.

I took three distinct approaches to this problem, two of which failed and one of which continues in use.

First, I keep a copy of each program ever submitted. As each new program was submitted, I did a compare with past work for that same task. If any programs were identical, I was notified.

Second, I customized many problems to individual students. George might be assigned to print the numbers from 1 to 10. Fred might be assigned to print the numbers from 1 to 100. George might have numbers right justified in a field of width 4. Fred might have numbers left justified. This added weight to plagiarism charges since the correct programs could not have been the same.

Third, I gave in-class programming tests several times each semester. The tests were closed-book and closed-notes, but consisted of simple problems (easy to write if you had been paying attention and doing your lab work instead of copying, and also quick to grade).

13.1 Comparing for Duplicates

There is a technological problem in comparing a new submission to a large body of prior work. I took several steps to make this managable. First, I restricted my attention to submissions for the same problem. Second, I restricted my attention to successful (final) submissions for the same problem. Third, I decided to calculate a checksum for each submission and store it in a database. If the checksum matched, then I would do the full check. The checksum proved to be the most efficient approach, reducing a order(n-squared) problem down to order(one).

At first, I got a lot of matches with previously submitted programs. Many of these could be explained by a few natural factors. (1) A similar program was in the textbook, so the student did a copy and modify. This was acceptable. (2) The teacher (myself) gave a similar program as an example in class. (3) The program had a naturally common solution. For instance, although there are many was to print the numbers from one to ten, this one is fairly popular.

```
for ( i = 1; i <= 10; i++ ) { print i }
```

I compensated by eliminating problems that had one typical answer. I also started reporting the authors of the programs that matched. If the instructor was on the list, or the tutor was, or lots of people had the same program, I ignored it. I focused in on cases where the same student A seemed to be copying from the same student B consistently.

Another problem that the reader may have noticed is the ease with which a student could make minor changes in a program before submitting it. I handled this by making several normalizations to the program before doing the comparison. First, I removed all white space and comments. Second, I removed all quoted strings. Third, I expanded `#define` statements (for C). This allowed me to catch a large number of questionable submissions, but it was still trivial to change a variable name or reverse two lines of the program. The ease of making such changes made it discouraging to try to compensate for all of them. On the other hand, the students most likely to cheat by plagiarism are also the ones most likely to do a bad job of it; those clever enough to cover their tracks are probably clever enough to do their own work.

I actually identified, documented, and prosecuted several cases before the Honor Code committed for plagiarism. But it took a lot of effort for marginal returns and I got tired of it. I also discovered that students like to work together. It is a social thing for many of them even though it was not for me. This led to lots of false positives. Eventually the key thing was whether they learned the material. As this was reflected on the exams (I called them midterms), I decided to ease up on the cheating.

13.2 Customized Tasks

The customization of tasks to individual students is still working and used for some problem sets, but I do not feel that it is producing any powerful results. I have left it alone and not added this feature to any new assignments.

13.3 Exams

I use my exams largely as a verification that the students can perform at a level indicated by their lab work. Generally I can grade an individual problem in ten seconds or less because of their simplicity. Here is an example of a typical problem from the first midterm in the Programming II class.

Sample Problem: Read lines from STDIN until you get a blank line. On each line is a number (e.g., 13 or 98.6). There will be at least one number. Report (a) how many numbers were read, (b) what is their total, (c) what is their average. Do not use any kind of array. Use a small, constant amount of storage.

There was still trouble, however. I had a few students who did incredibly well on the lab assignments, but failed miserably on the exams. I followed up on one outstanding case (scoring 90% or better overall, but scoring about 10% on the final) and was told by the tutors that this individual would badger people for help until they gave him a bit. Eventually he would wheedle a

whole program out of his unwilling accomplices. But he could not perform on the test.

Because of this and similar problems with a few students, I instituted a grading policy that required a certain final exam grade to earn a certain overall course grade. For instance, to earn an A in the class, you must earn at least a B on the final. To earn a D- in the class, you must earn at least a 20% on the final. I have been very pleased with this approach and it has allowed me to worry less about the possible excessive teamwork outside of class.

14 GradeBot 2.0: Interactive

A few weeks after GradeBot 1.0 was unleashed on an unsuspecting student body, one clever student succeeded in submitting a program that contained an infinite loop. I had planned for this. Programs were only allowed to execute for two seconds before being terminated. However, this loop contained a print statement. 80 thousand lines of output later, his program had been terminated and the results were being emailed back to the student. Do you have any idea how long it takes to email 80 thousand lines? It took about an hour. I immediately installed a patch to remove duplicate lines (counting them of course). Later I restricted the email to 2n lines, where n is the number of lines that were expected. But these were stopgap measures.

The real solution came when I rewrote GradeBot in `expect`. Expect is a Tcl add-on that is optimized for dialogs between programs. You say this. I say that. You say this. I say that. I revised my entire suite of programs away from the STDIN / STDOUT model into a get/put dialog model.

Under the new model, GradeBot engaged the student program in a dialog. One line at a time, input was fed in and output was accepted back. Everything was logged for the user. At the first failure to get the expected output, the user's program is terminated and the expected output is logged together with a suitable error message.

Although I was not trying to solve the excessive output problem, it came as a delightful side effect. Another nice side effect was the almost complete disappearance of `core` files left over from crashed student programs.

There was a downside, though. The new model did not support "end of file" as an input to a program. Once the end of file was sent, `expect` refused to accept any further output from the student's program. As a result, programs which must be tested for end of file handling still use the batch STDIN/STDOUT features of GradeBot 1.0.

15 GradeBot 3.0: On the Web

Maybe this should be described as GradeBot 2.5 because the underlying engine did not change. The only thing that is new is a web interface for students.

The web interface was motivated by the fact that the first two weeks of each semester were taken up in teaching the students to use a programming editor (I used emacs, I allowed vi) and how to submit their work and retrieve their results. We got through it but the delayed gratification was frustrating. I wanted to be able to write programs the first day, or at least the second.

With the web interface, students entered their programs into a textbox in a browser. After pressing the submit button, a CGI program ran. This program, named WebBot, simply acted as a front end to GradeBot. It organized the submission and passed it along through traditional means (i.e., by email). GradeBot responded by email and WebBot displayed the results in the browser window.

The WebBot interface was implemented by a senior student and has been very popular with the students. Over time we have recognized some advantages and disadvantages to the web interface:

Advantage: The students can write programs on the first or second day.

Advantage: The students do not need to learn any special purpose commands for joining a class or seeing a report of their labs completed. These can all be handled through buttons on the web interface.

Disadvantage: Students are unable to invent and test their own programs. They are restricted to the small list of programs that GradeBot knows about.

Disadvantage: Students do not even begin to test their own programs. They type and press submit.

Disadvantage: Students do not become familiar with the command line interface or with a programming text editor.

Our evaluation of these advantages and disadvantages is that for Programming I, for the typical student, the advantages are compelling. The advanced student can be sent to the tutors to learn the command line interface and how to write and test his or her own programs. However, several weeks into Programming II we introduce the command line interface and programming editors and force the students to become familiar with them. We then allow students to use whichever interface they find most convenient for the task they have.

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.