

is common to follow the generic message with pasted code to illustrate exactly where the problem lies.

The grader can use the quit command to stop grading at any time. The grade files for the previously graded submissions will be retained. The next time the gradeall script is run, grading will begin with the first ungraded submission. To regrade an already graded submission again, the grader can remove the Grade/userid file and run the gradeall script.

### RETURNING GRADES

The sendgrades script will e-mail all the Grade/userid files to the respective userids with a subject of "Grade on lab labname" where labname is the name of the lab directory (used by the student in the original submission). The Grade/userid files are moved to a directory called Sent as they are mailed. This prevents sending multiple copies of messages and makes it easy for the grader to determine if something has been sent. These files are kept in the grader account for documentation.

### RECORDING GRADES

After the messages and grades have been sent to the student, the final need is to put the grades into a spreadsheet. The collectgrades script collects all the grades from the messages by extracting the line containing the grade and then picking off the grade. It builds a file containing a line for each student with the students userid, a tab, and the grade. Copy and Paste can be used to put the grades into a spreadsheet. No manual recording is required saving considerable time and reducing errors.

### CLEANUP

The process of grading creates numerous files. These include the executables, compiler messages, and program output for each student. A clean script is provided that cleans up all the easily reproduced files. It does not remove the comments or the messages that were sent to the students. For significantly more space savings, after the lab directory has been cleaned it can be tarcompressed. The nature of these files is such that tarcompress will significantly reduce the used space. This allows the lab to be retained through the end of the semester without using excessive resources.

### CONCLUSIONS

We have been using these programs and scripts for a year. They have been used by over half the faculty in our department with much praise. Having a very flexible submission-execution-grading system has greatly eased the difficulties of grading student programs.

The submission programs have been very reliable, even when e-mail was disabled for a week due to system problems. Of course, returning grades was delayed during that time.

This set of tools is vastly superior to submissions on paper or by e-mail because it simplifies every step. There is no action required to prepare each program for execution as would be necessary for e-mail submissions. There are guards against being sent executables rather than source (as always seems to happen with e-mail submissions). If the grader is so inclined, a copy of the student program may be modified to demonstrate to the student the effect of suggested changes. Some graders found they had more time to write individual comments to students when they did not need to retype general comments. Much time is also saved by the automation of repetitive tasks such as saving e-mailed programs, compilation and execution, and sending mail. Although this system has been used for programming assignments, it could also be used for other types of assignments that can be submitted on-line.

The source for Grasst is available through my home page on WWW at <http://cs.millersv.edu/~hutchens/hutchens.html>.

From the grader's perspective, the lab directory is filled with the students' files. For single file submissions, the file that is submitted is renamed to be the userid of the student with the required file extension. This simplifies the task of verifying which account the submission was from and eventually mailing grades back to the student. If the submission is a directory of files, a directory named by the students userid and an extension of .dir (to simplify processing) is created. The student's files are placed in this directory with the original extensions.

## EXECUTION

When the grader is ready to run the program, data must be supplied. This data will be provided to each program as its standard input. To supply data, simply create a file called Data in the lab directory. If the program needs to be executed more than once, make a directory called Data and put each input file in that directory.

The runall script will compile each student's program and run it on the provided data. The compiler messages and program output are kept for each student. This process takes a few minutes and can be left to run by itself.

It is not necessary to have all the submissions to use runall. Any submissions that have not been graded will be run. It is also possible to provide command line arguments to runall to specify which student submissions are to be run. Hence, it is possible to grade incrementally as the submissions arrive. Of course, a resubmission by a student will require a regrading so this is not normally done.

The runall script is controlled by several environment variables that specify the extension, compiler, and run time limits (to break out of infinite loops). These are usually set in the .login script for the grader account since they are seldom changed from one lab to the next. If no values are set, defaults are used. The values are printed when runall starts.

The script automatically handles single files or directories of files.

## GRADING

After runall has compiled and executed each submission, the grading process can take place. The gradeall script automates much of the mundane activity here. It is assumed that a scrolling terminal window is being used since each submission is usually a good bit more than one screen full of information. Each submission that has not yet been graded is handled in turn.

First the source is written to the screen. Then the output of the compiler (warning and error messages) is written. This is followed by a copy of the supplied input data and the output that was produced.

After this information is in the window, the script loops asking for commands or comments.

The available commands are:

- e: edit the message file for this student
- i: list the comments that are currently included for this student
- l: list the already named comments for this lab
- q: exit the program without saving any comments for this student
- s: save the comments for this student and exit the loop

If the entry is not a command, the entry is taken as the name of a comment to be included. The name is used as a file name in the Replies directory. If the file already exists, it is included in the student's message file and another command or comment is requested. If the named comment does not exist, the user's specified editor (EDITOR environment variable) is invoked on a new file with that name. The desired comment is typed into the file and saved. Once saved, it will be included in the student's message file. If no file is saved, nothing is added.

This process allows the grader to write a message for a given problem once, and then include it by name for all subsequent students. The message might include a statement of the problem, the number of points lost, suggested corrections, or suggested code.

Normally, the messages are named with a numeric extension. After the comments are finished and the s(ave) command is issued, these extensions are added together and subtracted from the total points for the lab (the totalpoints environment variable; default 100). The final score is calculated automatically. The final score is displayed and the grader can accept it (with return) or enter an alternate score.

The messages for a student together with the final grade (and optionally the program and the input/output pairs) are placed in a file called Grade/userid and can be mailed to the student using the sendgrades script.

The e(dit) command allows easy inclusion of sections of code from the student's program. Since they are already in the window, they can be copied and pasted into the editor. It

These utilities expect a standard file system structure in the /grader/graderaccount directory (see Figure 1). To set up the basic requirements, while logged onto the grader account, run the script submitinit. This script will create a directory called /grader/graderaccount/submissions, set the /grader/graderaccount directory as readable by everyone and set the /grader/graderaccount/submissions directory as readable only by owner. This assures that the submit program can determine if the account is accepting submissions, but the actual submissions will not be available for students to copy them.

The submitinit script will also place a copy of the acceptsubmit program in the /grader/graderaccount directory and set it to run setuid owner. This program is run by the submit program to copy the submission from the student's account to the grader account (see the section on submitting).

To simplify things for the students, the acceptsubmit program should be removed from this directory if it is not going to be used for a while (as in when that professor is not currently teaching that class). This will remove the account from the list of accounts accepting submissions that is printed as an aid to the student by the submit program. The submitinit script may be run again to re-enable submissions.

The next step in preparing the grader account is to set up a directory for each assignment or lab that the students will submit. This directory is a subdirectory of submissions. For example, to allow for the submission of a queue lab, create a directory called /grader/graderaccount/submissions/queue.

Each subdirectory of the submissions directory should correspond to an assignment that is currently outstanding. When the professor wishes to stop accepting submissions for an assignment, the corresponding directory is moved out of the submissions directory. I personally maintain tograde and graded directories and move from submissions to tograde when the submission period is over and from tograde to graded after the assignment has been graded.

The last task in preparation is to decide what file types (extensions) will be accepted. The default behavior is to accept a single file that ends with a .p extension. To choose a different extension, create a .submitdefaults file in the /grader/graderaccount directory or in the specific lab directory. The acceptsubmit program reads these two files in turn (if each exists) so that the grader account can have a master .submitdefaults file and override it for specific labs. The .submitdefaults file should contain a line such as:

```
filetype = .cc
```

This will accept C++ files rather than Pascal files. If the students are building programs using more than one file, a directory of information must be copied. To do this, use:

```
filetype = dir:.cc,.h
```

This will accept all files in the current directory that end with .cc or .h. This does not copy executables or other files. This also has the effect of forcing students to use a unique subdirectory for each project that requires multiple source files.

The current version does not permit the submission of multiple directories of files as would be needed for a large semester project with subprojects in separate directories.

It is possible to accept every file in the directory with:

```
filetype = dir,;
```

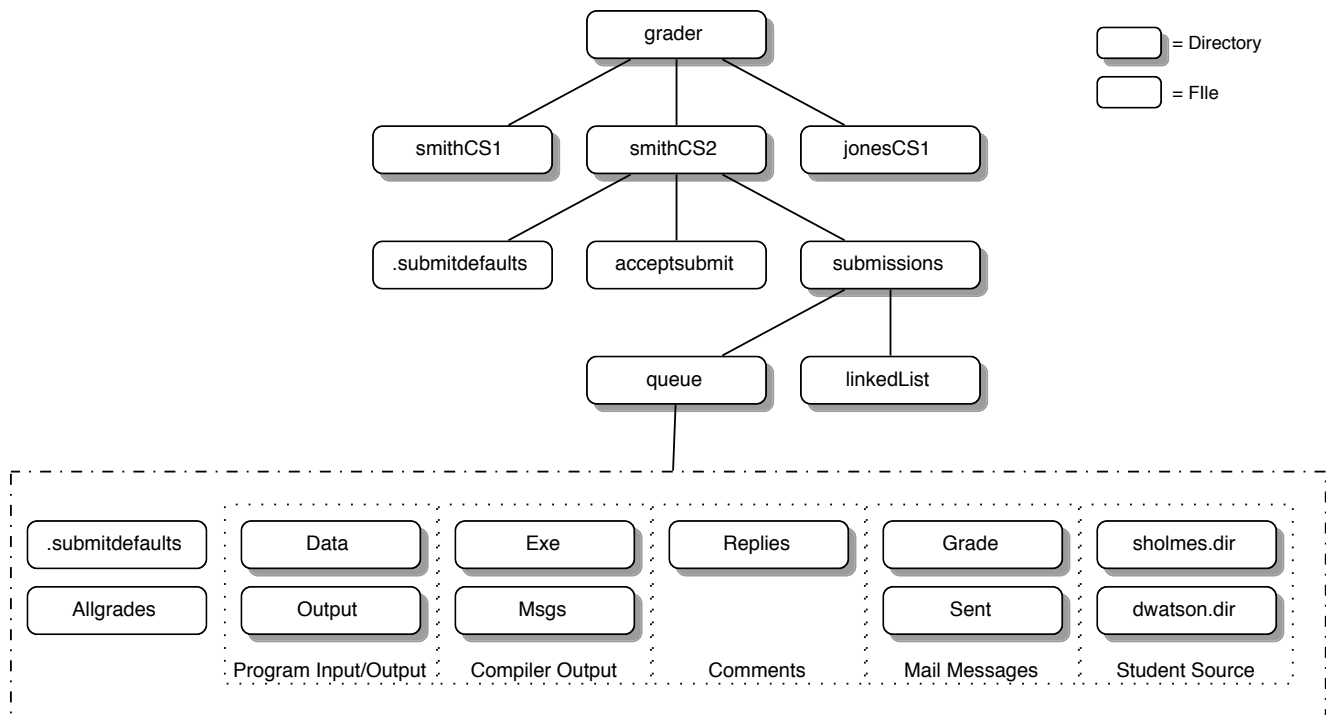
Setting the appropriate file types greatly decreases the probability that a student will submit the incorrect file. For example, a common problem of e-mail based submissions is the accidental submission of the executable program rather than the source code. That is not likely under this system. Of course, no system is foolproof. Any student can, with some effort, submit whatever file desired by changing the file name to include the appropriate extension.

## SUBMISSION

From the student's perspective, submitting is extremely easy (and hence not prone to error). In a shell, with the current working directory at the place where the compiler might be invoked, the student types submit.

The program responds with a list of accounts that are accepting submissions. The student selects one by typing the userid of the grader account. The program then gives a list of labs. The student selects one. The program then gives a list of files that may be submitted (this is not done when a directory is expected, as all appropriate available files will be submitted). The student selects the file. Finally the program repeats all information and asks for confirmation. If the student responds yes, the file(s) are copied into the lab directory of the grader account and named as required for further processing. If a question has only one answer, submit assumes it is the correct answer and the question is not asked. Confirmation before final submission is always required.

A student may submit a single lab multiple times as long as the lab is available for submissions. Each submission is maintained in the lab directory of the grader account. Only the last submission is used by the grading scripts.



**Figure 1: Directory Structure**

```

The currently active grader accounts are:
jonesCS1 smithCS1 smithCS2
Please enter name of grader account: smithCS2
  
```

```

The lab choices are:
queue linkedList
Please enter lab name: queue
  
```

```

Files: driver.cc queue.cc queue.h
will be submitted to grader smithCS2 as lab queue.
Is this OK (y,n)? y
  
```

```

Files: driver.cc queue.cc queue.h
have been submitted to smithCS2.
  
```

The queue assignment is due, and all the students have submitted their solutions. Smith moves the queue directory out of submissions, opens the queue directory, and sees that there is an sholmes.dir directory along with others for each student. He creates input data in a Data directory. He types a single runall command to compile all the students' solutions and execute them with the input data.

He types gradeall, sees the first student's program and any compiler messages. This is followed by a copy of the input data and output. He notes that the student didn't handle the empty file appropriately and should lose ten points. He types empty.10, automatically enters the editor, types a

message about handling empty input, and saves the file. After reading the program and adding other comments, he saves the messages to this student. Grasst suggests a grade of 75 based on the points from the messages. He accepts that as the grade for this student and goes on to the next student. He notes that the next student also mishandles the empty file. He types empty.10 again, and the empty.10 message is added to the student's mail message without Smith retyping it.

After grading each student's solution, Smith runs sendgrades to e-mail the messages to the students. He runs collectgrades to gather a list of grades to paste into his grade spreadsheet. Finally, he runs clean to remove extra files created by Grasst.

## PREPARATION

Before students submit assignments, an account must be prepared to receive them. The account must have a directory inside the /grader file system that is mounted from the file server on the student's workstation. This directory is named with the userid of the account. We use separate grader accounts for each professor-class combination and /grader contains the home directories for those accounts. The submit program looks in the /grader directory to find accounts that are accepting submissions.

# GRASST: A SYSTEM TO ORGANIZE AND SUPPORT THE GRADING PROCESS

David H. Hutchens

Department of Computer Science, Millersville University, Millersville, PA 17551  
hutchens@cs.millersv.edu

## ABSTRACT

This paper describes Grasst, a grading assistant, that consists of a group of UNIX programs and shell scripts to assist in grading assignments. Students submit assignments for a given lab using a *submit* program. The grader uses the *runall* script to compile and run the submissions against grader-provided data. The *gradeall* script displays each submission with its runtime results and allows the grader to include comments. The comments may be coded to calculate a suggested grade and may be customized on-the-fly. Commonly used comments may be repeated for other students by name rather than retyping. The *sendgrades* script mails the grades and comments to the students. Finally, the *collectgrades* script creates a list of grades for a grade book or spreadsheet. The scripts are portable, easily customized, and easy for students and professors to use.

## INTRODUCTION

Every instructor of programming has faced the problem of receiving and grading programs. Several approaches have been tried over the years. Programs were traditionally submitted on paper. Sometimes these submissions included the results of compilation and execution on a data set chosen by either the instructor or the student. In either event, the student knew what the data set would be and seldom tested the program with any other data.

One way to avoid this is to have the student submit the program electronically. This has been done through two major techniques. The first is to have the student submit the assignment on diskette. Grading these submissions required a large amount of diskette shuffling and the execution of numerous commands. Alternatively, some instructors have tried submission by e-mail. This avoids the diskette shuffle but many problems remain. The program must be extracted from the mail message and named prior to compilation. Some mail systems do not provide for inclusion of files in

the messages and require that the program be sent as normal message text. Other mail systems allow the inclusion of a program file directly (e.g., MIME mail). In either case, instructors perform many actions for each program before execution. Again, the program must be named manually, either at the time of extraction or at the time of execution.

Finally, after the compilation, execution, and grading, the grade and comments must be returned to the student. Without paper, there is a need to electronically return a response. This requires having the student's e-mail address available and typing or pasting it into the message.

Program evaluation is indeed a tedious task. There is a better way.

This paper describes a collection of utilities that greatly simplifies the mechanical portions of this task. These utilities do not provide for automated program evaluation other than compilation and execution. All programs are still read for style and other factors by the professor or grading assistant. It would be easy to modify the scripts to process the source through other tools as desired.

The next section provides a short example of using the system. Each of the following sections describes in detail the tools for each step of the process.

## EXAMPLE

Suppose Professor Smith is teaching CS1 and CS2 and has grader accounts of smithCS1 and smithCS2. From each account, he runs the *submitinit* script which creates a copy of the *acceptsubmit* program as well as a submissions directory. In smithCS2, Smith creates subdirectories inside submissions for two assignments—*queue* and *linked list*.

Figure 1 shows a typical directory setup including both the directories that are made by the user and the directories that are automatically generated by the utilities.

Student S. Holmes has completed the *queue* assignment and is ready to turn it in. He is already in the directory containing his program. He has the following interaction when he types *submit* at a shell command prompt: