A multi-national, multi-institutional study of assessment of programming skills of first-year CS students

Report by the ITiCSE 2001 Working Group on Assessment of Programming Skills of First-year CS Students

Michael McCracken (chair)
Georgia Institute of Technology, USA
mike@cc.gatech.edu

Yifat Ben-David Kolikant
Weizmann Institute of Science, Israel
ntifat@wisemail.weizmann.ac.il

Vicki Almstrum
University of Texas at Austin, USA
almstrum@cs.utexas.edu

Cary Laxer
Rose-Hulman Institute of Technology, USA
Cary.Laxer@rose-hulman.edu

Danny Diaz
Georgia Institute of Technology, USA
ddiaz@cc.gatech.edu

Lynda Thomas
University of Wales, Aberystwyth, UK
ltt@aber.ac.uk

Mark Guzdial
Georgia Institute of Technology, USA
mike@cc.gatech.edu

Ian Utting
University of Kent, UK
I.A.Utting@ukc.ac.uk

Dianne Hagan
Monash University, Australia
Dianne.Hagan@infotech.monash.edu.au

Tadeusz Wilusz
Cracow University of Economics, Poland
eiwilusz@cyf-kr.edu.pl

ABSTRACT

In computer science, an expected outcome of a student's education is programming skill. This working group investigated the programming competency students have as they complete their first one or two courses in computer science. In order to explore options for assessing students, the working group developed a trial assessment of whether students can program. The underlying goal of this work was to initiate dialog in the Computer Science community on how to develop these types of assessments. Several universities participated in our trial assessment and the disappointing results suggest that many students do not know how to program at the conclusion of their introductory courses. For a combined sample of 216 students from four universities, the average score was 22.89 out of 110 points on the general evaluation criteria developed for this study. From this trial assessment we developed a framework of expectations for first-year courses and suggestions for further work to develop more comprehensive assessments.

KEYWORDS

INTRODUCTION

Programming is one of many skills that computer science students are expected to master. In addition, most science, mathematics, engineering, and technology (SMET) programs expect that their students will acquire programming skills as a part of their education. The question is whether these requirements are being met. Are the appropriate assessment measures in place to determine if the students have acquired the necessary programming skills? We think not, but wanted to gather evidence that would confirm or refute our observations.

This working group arose from concerns expressed by many computer science educators about their students' lack of programming skills. Quite often these concerns were focused on basic mastery of fundamental skills of programming. A study by [8] identified similar deficiencies in programming skill, although their study focused on the teaching of programming. In several other studies that have considered issues of learning to program, assessment has been a part of their methodology. For example, [6] studied students learning Basic; [7] looked at conceptual "bugs" of novice programmers; and [9] studied novice programmers' misconceptions. While the results from these studies can help computer science educators improve the teaching of programming, they do not answer this question: Do students in introductory computing courses know how to program at the expected skill level? This working group collected data from several universities and found that the students' level of skill was not commensurate with their instructors' expectations.

Two issues are central to our effort:

- Learning to program is a key objective in most introductory computing courses, yet many computing educators have voiced concern over whether their students are learning the necessary programming skills in those courses.
- The development of CC2001 [1] represents the next evolutionary cycle of the requirements for computing education. These requirements are slated to become the new standard for computer science education and will form the basis for accreditation of computer science programs in the USA. The requirements for introductory computing courses in the ironman version of the CC2001 prescribes the set of expected programming skills students should acquire but includes little information on assessment. The efforts of this working group may contribute to developing assessments for use by CC2001 implementers.

The remainder of this report is organized into eight major sections. We begin by describing a framework for learning objectives during the first year of computing courses. The next section explores a variety of assessment approaches and motivates the choice we made for this study. Next we describe the methodology for the trial assessment, including the work we did in the months before the ITiCSE conference. In the analysis section, we describe what we gleaned from the data during our working group's meetings at the conference. The remaining sections interpret the results, discuss implications and possibilities for further analysis, raise issues to be addressed in follow-on studies, and propose a model for driving this work further.

A FRAMEWORK FOR FIRST-YEAR LEARNING OBJECTIVES

When faced with understanding student performance, a natural question is "What should be assessed?" The working group discussed these issues and developed a framework of first-year learning objectives, both to clarify what we expected students to have learned during their first year and to allow us to evaluate how well the instruments for this study assessed the learning objectives.

For first-year computing students, a fairly universal expectation is that they should learn the process of solving problems in the domain of computer science, in order to produce compilable, executable programs that are correct and in the appropriate form. As the framework for the learning objectives of the first year, we expect computing students to learn to successfully follow these steps:

1. Abstract the problem from its description
2. Generate sub-problems
3. Transform sub-problems into sub-solutions
4. Re-compose the sub-solutions into a working program
5. Evaluate and iterate

In general, all Computer Science programmes aim to produce students who can reliably follow these steps in solving discipline-specific problems, independent of the particular programming paradigm being used. This also remains as a (possibly implicit) goal as students progress through their programmes, although the domain of application, as well as the scale and complexity of problems addressed, changes. The following clarifies what is involved in each of these problem-solving steps.

1. Abstract the problem from its description — First-year assessment exercises are generally framed in terms of a concrete, usually informal, specification of a problem for which students are required to implement a solution. Starting from this specification, students must first identify the relevant aspects of the problem statement. Next, students must model those elements in an appropriate abstraction framework, which is probably predetermined based on the approach being used in the solution space (e.g., procedural, OO, functional, logic) and heavily influenced by the teaching approach.

2. Generate sub-problems — The scope and importance of this step in the problem-solving process may be dependent on the design approach adopted. A functional decomposition of a structured program often requires further decomposition. In an object-oriented solution, the previous step has probably designed the classes needed, although at this stage, there may be factorization of methods out of others already in the design.

3. Transform sub-problems into sub-solutions — Here, the student must decide on an implementation strategy for individual classes, procedures, functions, or modules, as well as on appropriate language constructs (solution representations). This includes deciding on data structures and programming techniques. A crucial aspect of this step is the implementation (and testing) of the sub-solutions. The solution should be correct and in the appropriate form, that is, it not only produces the right output but is also modularized, generalized, and conforms to standards. Some language constructs may be inappropriate in particular domains or particular pedagogies; for example, it is not possible to use recursion in all languages. This step is typically the first point in the process at which significant involvement with tools (e.g. a compiler) is possible.

4. Re-compose — In this step, the student must take the sub-solutions and put them back together to generate the solution to the problem. This step probably involves creating an algorithm that controls the sequence of events.

5. Evaluate and iterate — Finally, the student must determine whether the earlier steps in the process have resulted in a good solution to the problem and take appropriate action if not. The solution must be tested thoroughly, and some of the earlier steps may be revisited if the solution fails any tests. The solution must be debugged to correct runtime and logic errors.

While the above framework of learning objectives represents an ideal and generalized situation, there are some problems with this abstraction. Particular pedagogic approaches and tool-chain support might change details of the sequence. For instance, an approach based on extreme programming (XP) [2] would make the testing activity much more central, so work on that aspect would begin much earlier in the process. The availability of tools such as BlueJ [3] would enable testing to be performed more easily at step 3, rather than waiting until step 5. Use of design tools and notations can encourage students to check submissions at an earlier stage in the process. Whatever the variations, however, all of the steps in the process should still take place.

ASSESSMENT INSTRUMENTS FOR FIRST-YEAR CS

This section reviews general requirements for assessment and describes types of assessment frequently used in first-year computing courses. In reviewing these strategies, we discuss how well each meets the general requirements for assessment. We emphasize that assessment must be tied to the educational objectives discussed in the preceding section on the learning objectives framework. We conclude this section by evaluating how well the trial assessment met these assessment requirements.

Two main categories of assessment are *objective testing* and *performance-based assessment*. Objective forms of assessment, such as multiple-choice questions, can provide a cost-effective means for determining student knowledge about areas such as language syntax or program behavior. Objective testing can provide instant feedback and can be used for both formative and summative assessment. On the other hand, multiple-choice questions cannot directly test students' ability to create working computer programs.

In performance-based assessment, students are assessed for their ability to create programs. Criteria for performance-based assessments include: fairness, generalizability, cognitive complexity, content quality (depth) and coverage (breadth), meaningfulness, and cost [4,5]. Below, we present three common forms of performance-based assessment instruments and discuss how well they meet the learning objectives framework from the previous section, as well as the seven criteria given earlier in this paragraph.

1. *Take-home programming assignments*

   Typically a number of these assignments are given during a course. Such assignments tend to be fairly large scale with a fairly generous maximum timeframe set for completing them (up to several weeks). Such assignments tend to cover all five aspects of the learning objectives framework. They generally contain a large amount of cognitive complexity. They are fair, generalizable, and meaningful in the sense that students are operating in an environment that is close to reality; however, students are penalized if they are unable to spend enough time completing the assignment. This type of assessment is more vulnerable to plagiarism than are some of the other assessment approaches.

2. *Examinations* (short answer)

   These examinations (such as asking students to generate code fragments) can be used to assess all five learning objectives, although items on such examinations often tend to concentrate on steps 3 and 4 of the learning objectives framework (decomposition into sub-problems and transformation into sub-solutions). It is difficult (but not impossible) to make short-answer examinations meaningful or generalizable because of the limited time available for students to complete them, but they can provide cognitive complexity at low cost.

3. *Charettes* (the method used in this study)

   Charettes are short assignments, typically carried out during a fixed-length laboratory session that occurs on a regular basis. The closed nature of these sessions reduces the opportunity for plagiarism. Charettes provide coverage of the learning objectives framework, although in a manner that is more superficial and less cognitively complex than is possible with larger take-home assignments. The experience of completing a charette may not be as meaningful or generalizable as larger assignments. Charettes may be unfair to students who have test anxiety or troubles with time pressure.

Once an assessment instrument is chosen, the scoring criteria must be determined. One approach to scoring would be a raw assessment of whether the program works (although this is not particularly useful for formative assessment). It is common for first-year computing instructors to examine the source code and other written materials as part of their assessment strategy. Another approach to assessment is to combine one of the above with interviews in which the students describe their process and product and thus demonstrate that they understood what they have presented.

In this study, the form of assessment used was the charette, a short, lab-based assignment. We selected this assessment type to foster a fairly uniform environment across universities at a relatively low cost. Our charette provided fairness in the sense that all students were operating in a similar environment, although this approach can be seen as discriminatory against students with test-taking anxiety. The exercises did offer cognitive complexity and covered all parts of the learning objectives framework reasonably well. In the Methodology and Analysis sections, we explain the criteria we used in assessing the students' programs.

METHODOLOGY

To help determine the programming ability of first-year computing students, the working group developed a set of three related programming exercises that students at several universities would be asked to solve. The exercises, which varied in difficulty, were designed so that, theoretically, students in any type of Computer Science programme should be able to solve them. Students could use any programming language to implement their solutions; we assumed that they would use the language that they were required to use for the course they were taking at the time. Students would only have to complete one exercise of their instructor's choosing. The opinion of the working group's participating schools was that a student at the end of the first year of study should be able to solve the most difficult exercise of the three in about an hour and a half.

The exercises focused on arithmetic expression evaluation. The easiest of the three exercises (P1) required a computer program to evaluate a postfix expression. The second exercise (P2) required a computer program to evaluate an infix expression with no operator precedence (the operations were to be performed strictly left to right, with no parentheses present). The last exercise (P3) required a computer program to evaluate an infix expression with parenthesis precedence (operations were to be performed left to right, with parentheses forcing sub-expressions to be evaluated first). Each exercise stated that input tokens (numbers and operation symbols) would be separated by white space to ease the process of entering data. Infix expressions would contain only binary operations (`+`, `−`, `*`, `/`, `^`); postfix expressions could contain unary negation (~) as well. The exercises are described in Appendix A.

To enable the work of students from different universities under different instructors to be compared meaningfully, the working group developed the General Evaluation (GE) Criteria shown in Appendix B. The criteria considered whether a student's program could run without error, process several arithmetic expressions, produce correct results, and determine when expressions contained errors. These criteria were strictly execution-based. To assess the style component of the GE Criteria, the source code was inspected.

The Degree of Closeness (DoC) Criteria given in Appendix C provided a subjective evaluation of how close a student's source code was to a correct solution. Students at some of the universities were also asked to complete a questionnaire (see Appendix D) that gathered demographic information, programming background, and reactions to the task.

Instructors at four universities administered the trial assessment as a laboratory-based exercise in their respective courses. Two used the first exercise (P1, postfix evaluation), one used the second exercise (P2, infix evaluation with no parentheses), and one used all three exercises, administering a different exercise in each of three sections of the same course. Students had either 1 hour (at one university) or 1.5 hours (at three universities) to write a computer program to solve the exercise they were given using the language they were taught in their classes (which happened to be either Java or C++). When finished, students submitted their executable programs and printed copies of their source code for assessment. At one university, the exercise was set up as an examination required of all students, while at the other three universities, the participants were volunteers who received extra credit points.

The computer programs were evaluated using the criteria in Appendices B and C. The GE Criteria assess how accurately the students implemented their solutions, and thus concentrate on the last two learning objectives (re-composition into a working program and evaluation). The DoC Criteria assess the results of the abstraction process and thus enabled us to see how well the students met the first three learning objectives (abstraction, decomposition, and transformation into sub-solutions). In addition, the instructor who gave the exercise as an examination graded the programs in the traditional manner in order to be consistent with the grading criteria for the remainder of the course. Outcomes of the assessments were reported to the working group leader for tabulation and cross-institutional analysis.

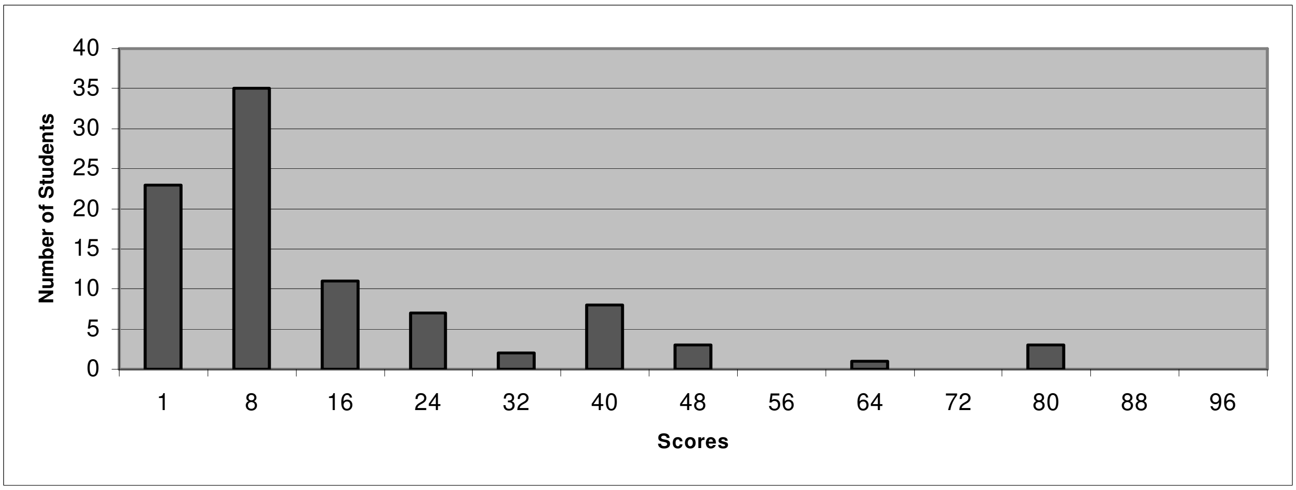

Figure 1: Distribution of GE scores on the combined P1 dataset (histogram)

ANALYSIS

Each instructor who administered the exercise applied the General Evaluation (GE) Criteria (Appendix B). All instructors produced an aggregate score for the General Evaluation Criteria; most instructors also reported the four component scores (execution, verification, validation, and style). In contrast, the DoC Criteria (Appendix C) were applied to the source code from all four universities by evaluators at a single university. The evaluators also generated comments to explain their reasons for giving each DoC score. In an informal inter-rater reliability test on scoring against the DoC Criteria, we found a high degree of correlation between evaluators.

Two of the four universities administered a local version of the Student Questionnaire (Appendix D). For all four universities, the exercise number (P1, P2, or P3) was recorded for each student as well as the programming language used (Java or C++ in all cases). The four participating universities were randomly assigned the codes School S, School T, School U, and School V. The instructor at School V reported a local grade on the exercise (which was given as an examination). We assigned each student an encoded student ID number in order to ensure anonymity.

Once the raw data from each university were entered and validated, the analysis followed two independent paths. One path was a quantitative analysis based on the GE score, the DoC score, and the other data available for each student. The second path was a subjective analysis that focused on several of the unsuccessful attempts to solve the assigned exercise, looking at comments embedded in the source code and information from the questionnaires. We present the outcomes of these analyses in the next three subsections.

Analysis of General Evaluation Score

The average General Evaluation (GE) score (combining the execution, verification, validation, and style components) for all students, all exercises, at all schools (n = 217) was 22.9 out of 110 (standard deviation 25.2). The scoring for each of P1 (Schools S, T, and V), P2 (Schools U and V), and P3 (School V only) appears in Table 1. Overall performance was generally fairly low.

	Average (stdev)
P1 (n = 117)	21.0 (24.2)
P2 (n = 77)	24.1 (27.7)
P3 (n = 23)	31.0 (20.9)

Table 1: GE average score by exercise

We assumed in this study that we would be able to safely combine data from multiple universities in our analyses. However, there are differences between the students at different universities (e.g., in raw talent, in previous experience, in courses completed), between how they are taught, in how the exercises were applied (e.g., examination grade vs. extra credit points, time allowed, hints given), and, especially, in how the GE Criteria were applied. We used a statistical test (Student's t-test) to compare the universities on each of the exercises. Schools S and T did *not* differ significantly on P1, but every other combination (Schools V and T on P1, Schools V and S on P1, Schools U and V on P2) did differ significantly ($p <$ 0.00001).

Table 2 summarizes the scores for each school across all the exercises. (Only School V used more than one exercise, P1, P2, and P3.) School V had considerably higher scores than the other universities. Note, however, that we cannot simply conclude that School V's students performed better; the differences may be due to factors such as how the GE Criteria were applied, what types of students participated, or how motivated students were to do well.

	Average (stdev)
School S (n = 73) — P1	14.0 (18.6)
School T (n = 21) — P1	12.0 (16.3)
School U (n = 47) — P2	8.9 (11.4)
School V (n = 23) — P1	*48.7 (25.7)*
School V (n = 30) — P2	*47.8 (29.1)*
School V (n = 23) — P3	*30.9 (20.9)*
Totals for School V on P1, P2, P3	43.0 (26.7)

Table 2: GE average score by university

GE Component *(and maximum score possible)*		**Average score (stdev)**	**As percentage of max score on component**
Execution	*(maximum: 30)*	7.2 (11.8)	23.9%
Verification	*(maximum: 60)*	1.6 (5.8)	2.8%
Validation	*(maximum: 10)*	0.3 (1.8)	3.2%
Style	*(maximum: 10)*	4.6 (3.4)	46.2%

Table 3: Average GE component scores and percentage of each component achieved

Schools S and T are not statistically different on P1, so we can combine those scores with more confidence that we can gain the benefits of an increased sample size and of describing students across multiple universities. On this combined P1 dataset (combining Schools S and T, n = 94) the average General Evaluation score is 14.0 (standard deviation 18.0). Figure 1 shows that the distribution of these scores is bi-modal. While the majority of the students did very poorly, there is a second "hump" in the distribution, indicating a set of students with somewhat better performance.

Bi-modal distributions ("two humps") appear throughout this data. Another example is the combined P2 dataset (combining Schools U and V), which has a similar bi-modal profile (Figure 2).

The majority of students working on P2 scored below 10 points and fewer than ten students earned between 10 and 35 points, while over thirty students scored between 36 and 54 points.

With such low scores, we were curious to know where the students lost points. The GE Criteria had four components: *execution* (did the program run?), *verification* (did it handle input correctly?), *validation* (is it the right *kind* of calculator?), and *style* (does it meet standards?). Though the scores are uniformly low, as a percentage of possible scores, students did best on the execution component (implying that, overall, they wrote programs that compiled and ran) and the style component (implying that the source code looked good). The lowest component scores were on the verification and validation components (Table 3).

ANALYSIS OF DoC SCORES

The Degree of Closeness (DoC) score, a five-point scale that rates how close a student's program is to being a working solution (see Appendix C), is particularly interesting to study because a single set of raters assigned the DoC scores for all four universities. Therefore, any differences in universities can be attributed to differences among the universities themselves, rather than to differences in applying the criteria.

We discovered that the GE and DoC Criteria *do* measure similar phenomena. The correlation between the GE score and the DoC score was significant (Pearson's r = 0.66).

The overall average DoC score (combining universities and exercises, n = 217) was 2.3 out of a possible 5 points

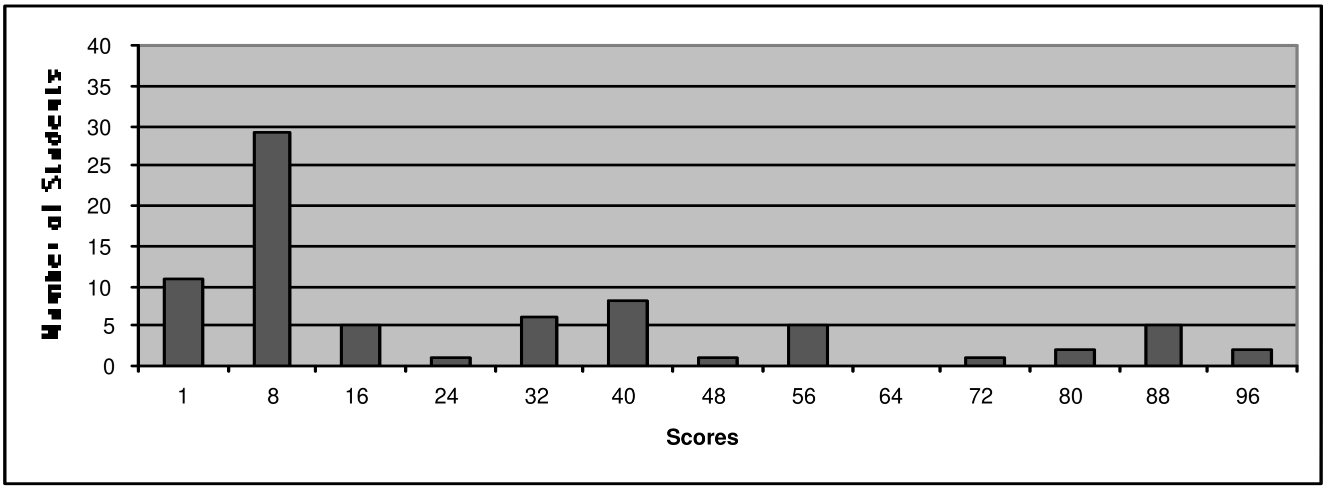

Figure 2: Distribution of GE scores on the P2 dataset (histogram)

(standard deviation 1.2). In general, student performance was low by measure of the DoC Criteria. The average DoC score for each exercise appears in Table 4. Students did best overall on the simple infix calculator exercise (P2), and next best on the RPN calculator (P1). This may be due to students' familiarity with infix calculators and notation and their lack of familiarity with RPN calculators, or perhaps due to mismatches between the demands of the exercise (e.g., stacks for RPN calculators) and the curriculum at a particular school.

	Average (stdev)
P1 (n = 118)	2.2 (1.2)
P2 (n = 77)	2.4 (1.2)
P3 (n = 23)	2.0 (0.9)

Table 4: DoC score by exercise

The distribution of DoC scores for the universities is shown in the first five rows of Table 5, with the average score for each university in the final row. School V had the highest DoC score, with School S second. The difference between universities is statistically significant (on a Student's t-test, $p < 0.01$).

At School T, we had the unusual circumstance of two different programming languages used in the exercises. About half of School T's students solved P1 using C++ (n = 10) and the rest solved the exercise using Java (n = 11). We calculated the average DoC score for each of these groups separately, then compared (using a Student's t-test) each group to a comparison group (School S's students) who solved P1 using Java. While School T's C++ programmers did significantly better than School T's Java programmers ($p < 0.001$), it is striking that the Java programmers at School T differ significantly from School S's Java programmers ($p < 0.001$), while School S's Java programmers and School T's C++ programmers do not differ significantly. Table 6 gives the average and standard deviation for each of these groups.

	Average (stdev)
School T's C++ Students (n = 10)	1.7 (0.8)
School T's Java Students (n = 11)	1.0 (0.0)
School S's Java Students (n = 73)	2.2 (1.1)

Table 6: Average score on P1 by School T's Java and C++ programmers and School S's students

QUALITATIVE ANALYSIS OF SELECTED SOLUTIONS

In our qualitative analysis of the data, our goal was to better understand some of the outcomes reported in the previous sections. We investigated the question "What went wrong?" (from both an instructor and a student point of view) for the students who produced an unsuccessful solution. The analysis was based on the students' source code as well as their responses to the Student Questionnaire (Appendix D). The analysis focused on students from Schools S and V whose DoC score was 1 or 2 and compared their performance with that of students at the same schools whose DoC score was 4 or 5.

First we investigated the data from the *instructor's* point of view to see how students were approaching the exercise. For the students whose DoC score was 4 or 5, we can say that little or nothing went wrong (i.e. they produced working solutions that really solved the exercise). These students can be characterized as individuals who figured out a solution for the exercise and either completed the exercise or were in the final phases of implementing a solution. In analyzing what went wrong for the students who earned a DoC score of 1, the results can be classified into three types:

Type 1 (null result): the student handed in an empty file.

Type 2 (unplanned result): the student's work showed no evidence of a plan to solve the problem. One explanation for this performance is that the student followed a heuristic in which they first did what they knew how to do, deferring the tasks about which they were uncertain, but were then unable to proceed beyond that point.

Type 3 (unimplemented plan): there is evidence that the student had a plan but did not carry it out. These students apparently understood what they needed to do and appeared to have a general structure for a solution. We further subdivide this type into two subtypes. For type 3a (unimplemented plan with promising approach), there was evidence that the student had identified a reasonable structure for solving the exercise. For type 3b (unimplemented plan with poor approach), the student apparently had a plan, but it was a poor one for the solution.

Next, we investigated the data from the *student's* point of view to better understand why the process of completing the exercise went so well for some students and so poorly for others. We contrasted student attribution of difficulties for students at School S whose DoC score was 1 with the attributions of students at the same school whose DoC score was 5. In the Student Questionnaire (Appendix D), students were asked to rank the difficulty of the exercise on the scale [easy, difficult, hard, impossible]. None of the School S students who earned a DoC score of 1 (n = 25) rated the exercise as easy. Six of these students did not respond to the questionnaire. Of the remaining nineteen students, six ranked the exercise as *difficult*, nine ranked the exercise as *hard*, and four ranked the exercise as *impossible* (and these were not necessarily the Type 1 students).

	School S	School T	School U	School V
Score of 5	3	0	0	9
Score of 4	5	0	2	15
Score of 3	22	2	11	19
Score of 2	18	3	15	18
Score of 1	25	16	19	15
n	*73*	*21*	*47*	*76*
Average (stdev)	*2.2 (1.1)*	*1.3 (0.7)*	*1.9 (0.9)*	*2.7 (1.2)*

Table 5: DoC score distribution by university

For the three School S students whose DoC score was 5, one thought the exercise was *easy*, one thought it was *difficult*, and one thought it was *hard*.

To gain some insights into why, we read the reflections reported by Type 1 students (null result) and students who earned a DoC score of 5. We found that the Type 1 students attributed blame for their difficulties to factors outside of their control. They blamed the amount of time available to solve the problem, their unfamiliarity with the computers in the lab, their lack of Java knowledge, and other external factors. None of the Type 1 students mentioned factors related to the process of solving the exercise. In contrast, students whose DoC score was 5 competently described the difficulties they experienced in the process of creating a solution. Many of these explanations illuminated particular aspects of the design phase or particularly challenging sub-problems. Examples of comments made by such students were “Simple errors got the best of me” (problem difficulty rated as *difficult*), “Could not solve for error case” (problem difficulty rated as *hard*), and “Implementation is wrong but easy” (problem difficulty rated as *easy*). Most of the students with DoC scores of 5 included comments in their source code that documented the cases for which the program did not work.

Due to the limited timeframe for the working group collaboration, this qualitative analysis is preliminary and incomplete. The Results section includes additional observations from the qualitative analysis and ideas for further qualitative analysis of this data, as suggested by the results to this point.

RESULTS

The first and most significant result was that the students did much more poorly than we expected. There are many possible causes: Our expectations may have been too high, the problems may have been too hard or a poor fit to the students’ background and interests, there may not have been enough time given, and so on.

We did answer the question we asked in the Introduction section: Do students in introductory computing courses know how to program at the expected skill level? The results from this trial assessment provide the answer “No!” and suggest that the problem is fairly universal. Many of the solutions would not compile due to syntax errors. This suggests that many students have not even acquired the technical skills needed for getting a program ready to run. While all the results were poor, School V’s students did significantly better than the other universities. Two important factors that may have contributed to this difference are: (1) The School V instructor had given the students an example to study, which was a complete answer to a similar problem, and (2) All students were required to take the exercise, which was given as an examination. Thus, sources of difference among the universities in this study could include type of preparation, motivation on this exercise (e.g., examination vs. extra credit), student characteristics (e.g. volunteers or compulsory participation), and issues such as curriculum and teaching style.

The School V instructor, who gave the exercise as an examination, applied local grading criteria in addition to the criteria defined for this trial assessment. We found that the correlation between the local grade and the General Evaluation score was high, but not overwhelming. One interpretation of this is that the two scores consider somewhat different features. It would be interesting to study these differences in order to gain a better understanding of the way instructors normally grade programming assignments and to contrast this with the criteria we used in this study. Local grades may consider more than performance on a single assignment. For example, a teacher may wish to reward effort or dramatic improvement, and there are certainly good reasons for doing so. Assessment in a study such as this one, however, considers performance at a particular instant. Give this difference in contexts, it is not surprising that the grade and the assessment score may differ.

We clearly misjudged the complexity of the exercises. The higher General Evaluation score of the students who worked on exercise P2 (infix notation without precedence) implied that this exercise was in some sense easier than exercise P1 (RPN notation). (Before conducting the study, we had rated P2 as being of “moderate” difficulty and P1 as being “simplest”). This points out more of what we still do not know about student learning and performance. P1 was undoubtedly difficult for students who had never studied stacks or other basic data structures.

The result about bi-modality is troubling. There are two distinct groups of performance in our datasets. This result suggests that our current teaching approach is leading to one kind of performance for one sizable group of students and another kind of performance for another sizable group. We need to keep in mind that different groups of students have different needs and strengths; we must ensure that the results from one group do not obscure our view of the other.

While the basis for comparison between programming languages is small for this trial assessment, we did unearth an interesting contrast. One school of thought says: "Java is better than C++ for education" or "Languages matter a lot—students learn better with X than Y." In this study, Java programmers from School S resembled C++ programmers from School T more than they resembled the Java programmers at School T. This suggests that the difference was not simply due to the programming language. Issues of how the course is taught and who the students are influence the outcome, rather than being simply a matter of programming language X vs. programming language Y. Future investigations must dig into how learning differs with different programming languages.

The fact that students did well on the style component of the General Evaluation Criteria indicates that students are responding to their instructors' admonishments about commenting and formatting of code. The other component scores (execution, verification, and validation) indicate that the code that students write does not meet specification; the only way to evaluate this is to run the students' code. An implication of this is the importance of actually *executing* student programs.

The significant number of solutions with a DoC score of 1 or 2 (i.e. students who were "clueless") raises the suspicion that those students need additional work during the first-year courses with developing skills in the first learning objective in our framework (abstracting the problem from a given description).

Many of the students who failed on this trial assessment had no idea how to solve the exercise. On the Student Questionnaire, the last question asked students: *What was the most difficult part of this assigned task? Was it the timed aspect of the problem, was the problem too difficult, etc.?* The following quotes are responses from students whose DoC score was 1 or 2:

- "I didn't have enough time"
- "I'm not good with stacks/queues."
- "Too cold environment, problem was too hard." [We believe the first phrase refers to the temperature in the physical setting.]

The most frequent student complaint was a lack of sufficient time to complete the exercise. This implies that these students could not accurately identify the main source of their difficulties in solving the exercise and therefore tended to attribute blame for their lack of success on factors other than themselves, such as a lack of time or the "cold" environment. In a multi-factor analysis, [11] found that attributing blame to external factors (such as "luck") was not uncommon, but was particularly hard to overcome. Once students attributed their failure to unstable factors that were out of their control, they rarely succeeded in future attempts.

One implication of this finding is that the implementation of first-year courses should make better use of available assessment methods and tools. Students should receive accurate feedback that allows them to become aware of their own limitations and difficulties—although such feedback alone will not necessarily convince a student that the reason he or she failed is at least partially internal rather than purely external.

Students often have the perception that the focus of their first-year courses is to learn the syntax of the target programming language. This perception can lead students to concentrate on implementation activities, rather than activities such as planning, design, or testing. Generally, this perception does not come directly from what their instructors are telling them and, in fact, this belief seems to be robust even in the face of instructors' statements to the contrary. Students often skip the early stages in the problem-solving process, perhaps because they see these steps as either difficult or unimportant. It is also possible that instruction has focused on the later stages, with an implicit assumption that the earlier stages are well understood or easy to understand.

The information from the students' reflections can provide useful information for improving the assessment process. The following two quotes are drawn from the responses to the same Student Questionnaire item as above by students whose DoC score was 2:

- "I had a plan, I did not know how to carry it out in Java."
- "The problem was too difficult, I lost a lot of time trying to understand how the computer work."

These quotes are from students who seemed to accurately identify their own difficulties and who took responsibility for their own performance. These students knew that they should go through a process of understanding, planning, and implementing. The earlier students' reflections give us little information about whether they were following these steps of problem-solving; in fact, the earlier students appear to have been lost and unable to point out what they do not know, blaming the environment or their poor understanding of a class of concepts.

The students' reflections provided useful information about the influence of the setting on student performance. Five School V students who earned a DoC score of 1 or 2 complained that they had a plan but could not handle the environment themselves and therefore could not translate their solution into a working computer program. When we interviewed the School V instructor, we learned that while the setting was indeed lab-based as specified in the instructions for how to administer the exercise, it was also the first time these students had taken a laboratory-based examination. This helps to explain why these students found it difficult to work on their own and performed rather poorly. Several students reported in the Student Questionnaire that stress played a major role in their unsuccessful performance, while others reported that they needed time just to figure out how a postfix calculator works. Being aware of such factors can help us as instructors to refine our assessment tools and give better guidelines on how to administer the tools. These data also give us insights into the students' performance that can be used to refine our approach to evaluating their knowledge.

DISCUSSION

In analyzing the data from universities in different countries, we have found that the problems we observed with programming skills seem to be independent of country and educational system. The most obvious similarity we observed was that the most difficult part for students seemed to be abstracting the problem to be solved from the exercise description. At all universities, the main student complaint was a lack of time to complete the exercise.

In this trial assessment, as in the "real world", it may be that black-box assessment of students' submissions reinforces students' views of implementation and syntax as the key focus of computer programming. Here we explore some possible reasons for the observed situation.

1. *Students may have inappropriate (bad) programming habits***.** When beginning their university studies, many students have prior experience in computer programming. Often students with such experience treat the source code as simple text rather than as an executable computer program that is supposed to accomplish a specific task. Their goal is simply to obtain a program that compiles cleanly; often they are then surprised by what the program really does when presented with data.
2. *Switching to modern (Java) object-oriented programming tools.* Anecdotal evidence and some research results (e.g. [10]) suggest that teaching an object-oriented approach to computer programming (for example, using a Java environment) requires more time before students have sufficient knowledge about the programming environment to solve problems on their own (which suggests that less time is required to achieve the needed level of familiarity with the environment in a procedural or functional approach). Therefore it is very likely that first-year courses using an object-oriented approach do not have room in the syllabus for fundamental data structures such as stacks, queues, and trees.
3. *Closed lab time constraint.* In terms of the way this trial assessment was administered, time pressure may have contributed to the poor results.

The qualitative analysis of selected solutions helped explain student performance and therefore highlights where future studies must improve over this trial assessment. One direction for further analysis would be to give a more in-depth characterization of the nature of student knowledge and difficulties within each DoC score (i.e. from 1 to 5). We could investigate this by considering the quality of the source code, the internal documentation, and the data from the Student Questionnaire. It would be useful to consider these issues from both from the instructor's point of view and the student's point of view. A student's reflections can provide important clues to whether the student understands his or her own limitations in knowledge. For example, the terminology that the student uses to describe his or her difficulties provides glimpses into the student's processes and problem-solving knowledge. These insights could help us better understand whether students are becoming competent in correctly identifying (and overcoming) their own difficulties.

In general, data analysis using qualitative approaches can provide information to help improve educational processes and refine assessment tools. For example, being aware of the factors revealed by qualitative analysis can assist us in developing better instructions for administering this trial assessment. The information generated by the qualitative analysis can also help make us aware of aspects of our students' behavior that we otherwise would not notice. Finally, the information from qualitative analysis can provide better and more accurate insights into what students know and how they use that knowledge.

To efficiently teach computer programming skills is difficult. The kinds of assessment that instructors use throughout their courses must provide appropriate information for understanding students' processes of developing programming skill. This trial assessment showed that most of the participating students failed to achieve one of the basic goals of a first-year computer science course: to acquire at least a basic level of skill with computer programming. This implies that it was the students' knowledge, rather than their skills, that enabled them to successfully complete their first-year courses. It is possible that either performance-based assessment tends to be improperly implemented or that it is often sacrificed in order to make assessment more objective.

ISSUES TO BE ADDRESSED IN FOLLOW-ON STUDIES

Several aspects of this study gave us cause for concern or raised points that must be addressed in future studies of this kind. These areas include the administration of the study, the exercises, and the challenges of multi-institutional collaboration.

Issues related to administration of the exercise

There are difficulties in comparing the performance of students with different programming backgrounds. In some universities, first-year students enter having already taken a general introduction to programming course, whereas in others most students are programming novices at the start of their first year of studies. Although some of the latter group may have prior programming experience from school, other universities, or self-learning, the preponderance of novices in the sample would affect the results from those universities. In future studies, we might specify the level of prior programming experience or the specific programming knowledge that the students are assumed to have for each exercise. It would then be fairer to allow instructors to choose the appropriate exercise to give to their students. The background questionnaire should also be modified to solicit information on students' prior programming knowledge.

Students were expected to solve the problem in whatever language they were learning in their course. As it happened, in our study all the students were learning either C++ or Java. The language of implementation affects the difficulty of the solution. For example, it is much easier to read data from a keyboard in C++ or even C than in Java. Many courses teach Java using classes supplied to simplify input from keyboard, but it was specifically stated in the instructions that students were not allowed to use such classes. The exercises should be chosen so that it is not necessary to use a technique that is clearly more difficult in one language than another.

These exercises were designed to be done using computers in a laboratory environment. The laboratory session must be monitored to ensure that nobody uses external means such as email or the Internet to obtain help with the solution. It was unclear from the trial assessment instructions whether the exercise could be done on an open-book basis. It was also unclear whether instructors were allowed to prepare the students for doing the exercise. Such issues should be explicitly addressed in the instructions in future collaborative assessment studies.

In some universities that participated in the study, the students were volunteers. In others, the exercise was compulsory. If students are asked to volunteer for a programming exercise, anyone who is weak in programming is likely to choose not to do it. This means that, in order to gain a true picture of the programming skills of students, the exercise must be compulsory for students. The only way to ensure that all students will attempt an exercise is to make its results count towards their final mark in a course. It must therefore fit into the assessment strategy of the course in which they are enrolled, as an examination for which a number of marks are allocated. In the future, it would help the analysis to record information about the conditions for each administration of the exercise, for example, *examination* vs. *extra credit* and *volunteers* vs. *compulsory*.

If the exercise is compulsory, a one-and-a-half hour laboratory consisting of only one question may be unfair. This is particularly true if this style of assessment is so different from what students have already done in their courses that they cannot determine where to start. An assessment of programming skill may need to take into account the fact that, in the "real world", a programmer usually does not have such a short time limit for understanding a problem and writing the required computer program. In addition, real-world programmers are generally free to refer to books and other resources if needed. Students whose primary language is not English may need a considerable amount of time to read the specification in order to understand what is required. In future studies, it may be necessary to allow students much more time than it is likely to take them to solve the problem. For example, if a teaching assistant can solve the problem in half an hour, it may be necessary to allow students up to three or four hours to solve it. Some students suffer from examination anxiety. To counter this, it would be possible to give students a week, say, to do the exercise, although this introduces more opportunities for plagiarism, and the assessment strategy would have to take this into account. Another approach would be to treat the topic area for the exercise as a case study that the instructor presents during one or more lectures. Basic materials for presenting the case study could be distributed to the participants. This would introduce some consistency in how the case study was introduced to students and could make it easier for students to quickly understand the requirements of the exercise in the closed-lab setting.

This study was not culturally neutral. For some universities, the exercises and instructions had to be translated into a language other than English. One way to minimize the effect of this difference would be to ensure a centralized translation to each language, which would ensure that all universities using a particular natural language use the same specification. Ideally, there should also be a validation step to ensure that the translated version of the exercise gives exactly the same specification as the original English version.

In future studies, instructors must receive sufficient notice of the study so that they have time to incorporate it into their assessment strategies for a particular semester. This point was a major factor in why additional universities did not participate in this trial assessment.

Issues related to the exercises

The exercises used in this study were probably discouraging for students with mathematical anxiety. Such students exist even in Computer Science programmes and are more likely to exist in other kinds of computing programmes that do not include compulsory mathematics courses or have strong mathematics prerequisites, such as a programme focused on commercial applications of computing. In future studies, a set of exercises of equivalent programming difficulty could be devised, and participating instructors could choose the most appropriate exercise for students in their programme. Alternatively, students could be allowed to choose the exercise that they felt most comfortable attempting.

The exercises in this assessment should have solutions that are unlikely to appear in the textbooks typically used by students in the first year. In this way, students who had used such textbooks would not be at an advantage over those who had not. To address this in future studies, a review panel, consisting of a representative sample of instructors, could be asked to provide feedback on the appropriateness of the task, the level students would need to be at to successfully solve the exercises, and whether they knew of any resources that would give some students an unfair advantage in solving any of the exercises. The review panel could include instructors from different countries, with different natural languages, teaching in different kinds of degree programmes, and using different programming languages.

In our study, the exercises were most easily solved using a procedural approach, and it was not easy for a student to decide which classes, attributes, and methods would be required if an object-oriented approach were taken. This may have confused many students. Given that most first-year programmes currently seem to be using an object-oriented language, the exercises should include options for which a natural solution can be designed using an object-oriented approach.

The specifications of the exercises in this study included details that were not relevant to the solution, which made it difficult for many students to achieve the first learning objective in our framework (abstracting the problem from the description). As stated earlier, many students (those with DoC scores of 1 or 2) did not get seem to get past that point in the problem-solving process. In the future, extra effort should be expended to make each specification as clear and simple as possible. One way to achieve this would be to ask the review panel mentioned earlier to suggest changes to the exercise descriptions, as well as to the instructions for administering the exercises.

Issues related to multi-institutional collaboration

This trial assessment is an example of collaboration on a single project across a variety of universities. Multi-institutional collaboration offers advantages as well as challenges. Among the advantages are an increased experience pool, a larger cumulative pool of students, and a wider variety of student profiles (increasing the potential for generalizability of results). At the same time, multi-institutional collaboration includes many challenges, some of which are addressed earlier in this section. Being separated physically makes it more difficult to coordinate protocols for conducting the exercises. It is also more difficult to make the data consistent (with respect to formats, field names, etc.) and complete (one university may collect data that is "lost" at another university, simply because the second instructor did not know to capture that information). Another important challenge is making the exercises sufficiently general so that they are neutral with respect to both culture and the university. Experience in this trial assessment suggests that we did not fully succeed in this. Our conclusion is that we must be cautious in defining general exercises, since we cannot assume that all first year programs cover the same material in content or emphasis, even within the boundaries of established curriculum standards and accreditation criteria.

Based on the experiences with this trial assessment, we offer the following advice for doing multi-institutional collaborations:

1. Appoint one research coordinator, who will be the main contact point for making decisions on the entire project. In our case, the WG leader was the research coordinator, who guided the entire process.
2. Do a trial run of the entire study, including analysis, in order to work out details of data formats and instruments.
3. Ensure that all source data can be traced to the interpreted data. For example, ensure that the printouts and files with the source code are marked in a way that associates each with the coded ID of the student who completed it.

CONTINUING THE QUEST

Because our preliminary work suggests that the problems we have observed are universal, the working group feels it is worthwhile to expand this trial assessment to include a broader base of computer science educators and universities. We envision establishing a central web site related to assessment of programming skills. Such a site could provide a gathering spot for links and materials related to this type of assessment, while at the same time being easily usable from throughout the world. The web site could include a registration process in order to allow restricted access to various parts of the assessment site.

The programming assessment site must support three main types of activities:

- **Assessment development**. The system should enable instructors throughout the world to participate in this collaborative project. For example, the web site should have features to support individuals who wish to submit new ideas or produce new assessments (perhaps following pre-defined templates obtained from the web site). The web site can also provide a technical forum where individuals developing assessment tools can discuss personal assessment experiences with others involved in the project.
- **Support for carrying out assessment and self-assessment**. This feature can serve two groups of users: students and instructors. The assessment web site can provide both groups of users with ready-to-use assessments and background information. As the instruments are filled out, the web site can collect the results and allow users to submit comments and feedback. Individual students would be able to use these tools for self-assessment and tracking personal progress. The assessment web site could also establish a worldwide database to accumulate information about students' computing knowledge and programming skills as measured by these assessments. Such a database would provide a basis for understanding student attributes within a single university, a single country, or even globally.
- **Communication environment**. While much of the information in the assessment web site will have strictly controlled access based on an individual's registered profile, the system could also allow the general public to access certain information about assessment. This would allow anyone interested in any aspect of assessing programming skills to exchange ideas and comments.

In order to realize the vision of an assessment web site, several organizational aspects are needed, including:

- a steering committee to guide the various efforts;
- a series of meetings, perhaps on an annual basis, where policy and structure can be defined;
- a committee devoted to maintaining the system; and
- one or more moderators who track day-to-day submissions from the public.

In order to foster interaction while establishing and building the assessment web site, a series of meetings could be held at regular intervals to gather individuals interested in contributing to this project. The meeting agenda would include developing the philosophy and strategy of assessment, accepting or rejecting proposed changes to the whole system, and managerial responsibilities such as designating the steering committee. It would make sense for the conference/workshop to take place in conjunction with a major conference such as the SIGCSE Technical Symposium or the ITiCSE Conference. The steering committee would be responsible for guiding the implementation strategy between the periodic meetings. The system maintenance group would be the professionals responsible for maintaining the system. Finally, the moderators would monitor the content of the system on a day-to-day basis.

The site with information from this working group is located at the URL:

http://www.cc.gatech.edu/projects/iticsewg/csas.html.

ACKNOWLEDGEMENTS

The chair of this working group thanks each member for her or his individual contributions. The members were what made this working group a success. This project required a great deal of dedication and effort by the members before, during and after the conference.

The group would also like to thank the organizers of the conference, Sally Fincher and Bruce Klein, and the working group leader, Roger Boyle, for giving us the opportunity to do this project. Finally, the group would like to thank Georgia Tech students Blake Markham and Prashanth Kolli, who helped with a lot of the logistics of the project.

REFERENCES

1. ACM & IEEE-CS Joint Task Force on Computing Curricula 2001 (2001). *Computing Curricula 2001, Ironman Draft*. Association for Computing Machinery and the Computer Society of the Institute of Electrical and Electronics Engineers. Available: http://www.acm.org/sigcse/cc2001 [2001, 5/16/01].
2. Beck, K. (2000). *Xtreme Programming Explained: Embrace the Change*. The XP Series, Addison-Wesley, 2000, Boston.
3. BlueJ (2001). *BlueJ, the Interactive Java Environment*. Available: http://www.bluej.org. [24 July 2001].
4. Hambleton, R.K. (1996). Advances in Assessment Models, Methods, and Practices. In D.C. Berliner and R.C. Calfee (Eds.) *Handbook of Educational Psychology*. New York: Simon & Schuster Macmillan.
5. Linn, R. L., Baker. E. L., and Dunbar, S. B. (1991). Complex, performance-based assessment: Expectations and validation criteria. *Educational Researcher*, *20*(8), pp. 15-21.
6. Mayer, R. E. (1981). A psychology of how novices learn computer programming. *Computing Surveys, 1*, pp. 121-141.
7. Pea, R. (1986). Language independent conceptual bugs in novice programming. *Educational Computing Research, 2*(1), pp. 25-36.
8. Soloway, E., Ehrlich, K., Bonar, J., & Greenspan, J. (1982). What do novices know about programming? In A. Badre and B. Shneiderman (Eds) *Directions in Human-Computer Interactions*, Norwood, NJ: Ablex, pp. 27-54.
9. Spohrer, J., & Soloway, E. (1986). Novice mistakes: Are the folk wisdoms correct? *Communications of the ACM, 29*(7), pp. 624-632.
10. Wiedenbeck, S., Ramalingam, V., Sarasamma, S. and Corritore, C.L. (1999). A comparison of the comprehension of object-oriented and procedural programs by novice programmers. *Interacting With Computers*. *11*(3), March, pp. 255-282.
11. Wilson, B. C., & Shrock, S. (2001). Contributing to success in an introductory computer science course: A study of twelve factors. In I. Russell (Ed.), *The Proceedings of the Thirty-second SIGCSE Technical Symposium on Computer Science Education*. In *SIGCSE Bulletin inroads*. *33*(1). pp. 184-188

APPENDICES

The information given in these appendices reflects updates made after completing the trial assessment. Some changes were introduced to clarify issues and to complete points that were missed during the initial development. The original and modified versions of the exercises and the instruments are available via the working group's web site at the URL http://www.cc.gatech.edu/projects/iticsewg/csas.html.

Appendix A. Overview of the Exercises

The content of three exercises developed for use in this study was distributed electronically to the participating instructors so they could easily cut and paste the text in creating their local versions of the assignment. As a baseline for difficulty levels, we hypothesized that second-semester computing students should be able to do the most difficult exercise of the three, Exercise #3, in 1.5 hours. To improve consistency, participating instructors received the following guidelines for how to administer the task.

- The students should work individually in a closed lab setting (proctored, with all work completed in the allotted time).
- The student's goal is to produce a working and tested program in the time allotted.
- This is a programming exercise, so students should produce a computer program. Any design

documentation, though important to solving the problem, is not important to this assessment.

The three exercises, referred to in the body of the paper as P1, P2, and P3, were as follows:

- Exercise #1 (P1): Programming an RPN calculator; difficulty level: 1 *(simplest)*
- Exercise #2 (P2): Programming an "infix" calculator without precedence; difficulty level: 2 *(moderate difficulty)*
- Exercise #3 (P3): Programming an "infix" calculator with simple precedence (i.e. precedence determined by parentheses only; no consideration given to operator precedence);; difficulty level: 3 *(most challenging)*

The exercise description included a common introduction for all three exercises. We suggested that students would need ten minutes to read and understand this background information. The main ideas in the introduction were:

- An explanation of the two main notations for hand-held calculators: Reverse Polish Notation (RPN) (also known as "postfix", which is generally used by Hewlett Packard calculators) and "infix" (which is generally used by Texas Instruments calculators).
- A description of how "post-fix" and "in-fix" expressions should be processed.
- A discussion of why RPN is simpler to implement (i.e. no precedence issues) while at the same time it is less intuitive for most users.

The individual descriptions of the three exercises provided the following information:

- User input is to come from the terminal's standard input; output should be directed to standard output for the terminal.
- The solution can utilize standard library routines provided by the language; no proprietary or other such libraries may be used.
- The operations that the particular calculator can process include addition, subtraction, multiplication, division, the power operator, and the inverse, or negation, operator. The "infix" calculator with precedence (Exercise #3) also included parenthesis pairs, which are used to indicate simple precedence.
- The description of each calculator shows the relative format for a line of input. For all of the calculators, some form of white space will delimit tokens (numbers and operators).
- User input will be entered non-interactively (so that the program is not allowed to query the user for additional information once the expression is entered), with the exception of the prompt to solicit the next line of input.
- The program should terminate when the input contains only the letter 'q'.
- When an error is detected in the input, the program should output an informative message and allow the user to begin entering a new expression.
- At the end of each calculation, the calculator should be cleared so the data structure containing the intermediate results is empty and ready for processing a new expression.
- Floating point arithmetic should be assumed and the program should allow non-integer expressions as valid input.
- Through several lines of a sample session, the description demonstrates a number of expressions and the results from the associated calculations for the specific calculator.

Appendix B. General Evaluation Criteria

Because this was a programming exercise intended to evaluate the programming skills of the participants, the evaluation focused on skills. The General Evaluation Criteria were designed to give reasonably consistent evaluations while allowing the participating instructors to still follow their normal grading process.

The total number of marks that a particular program could earn was 110. In the following, we have listed the allocation of marks immediately after each item. The style section was optional, since some instructors do have not style requirements in their introductory classes.

Execution *(30 marks)* – Does the program execute without error in its initial form? Does it compile without error? Does the program run successfully (no core dump or equivalent failure)?

Verification *(total of 60 marks, as broken down in the itemized list)* – Does the program correctly produce answers to the benchmark data set? This includes the following issues:

- *(10 marks)* The program should allow for multiple inputs of different arithmetic expressions (i.e., it should clear out the data structure properly between different expressions).
- *(10 marks)* The program should terminate correctly (i.e., entering the quit command should terminate the program).
    - *(30 marks)* The program should correctly process data sets containing expressions typically evaluated with a calculator. (Some sample expressions were provided to the instructors. The samples were not meant to be exhaustive, but to provide a benchmark.)
- *(10 marks)* The program should react properly to erroneous inputs.

Validation *(10 marks)* – Does the program represent the calculator type asked for in the exercise specification?

Style *(10 marks)* – Does the style of the program conform to local standards, including naming conventions and indentation? (The style measure was optional.)

Appendix C. DoC Evaluation Criteria

As a more subjective measure of the quality of a solution, the working group developed an indicator that we came to call the DoC score, for "Degree of Closeness" (or, with tongues firmly in cheeks, "Depth of Cluelessness"). The DoC score applies to programs that did not work and indicates how close the solution was to working.

To assign the DoC score for a student's program, the evaluator inspected the source code. The scores ranged from 5 to 1, with 5 being the best. Generally, the evaluators added notes to explain the reasons for the assigned score.

DoC Score	**Interpretation**
5	Touchdown. The program should have compiled and worked. If it did not work, it could be that the student simply ran out of time.
4	Close but something missing. While the basic structure and functionality is apparent in the source code, the program is incomplete in some way. For example, it might have been missing a method or a part of a method, but everything else seemed fine.
3	Close but far away. In reading the source code, the outline of a viable solution was apparent, including meaningful comments, stub code, or a good start on the code.
2	Close but even farther away. The outline, comments, and stub code showed that the student had some idea about what was needed, but completed very little of the program.
1	Not even close. The source code shows that the student had no idea about how to approach the problem.

Appendix D. Student Questionnaire

This version of the questionnaire was used at an American university. This questionnaire must be customized for each participating university to solicit equivalent information.

Part 1: Personal Information
Name: ______________________________ IDNUM: ____________________

(please circle the correct choices below)

Sex: **Male** **Female**

Class Rank: **Freshman Sophomore Junior Senior**

Overall GPA**:** **<2.0** **2.0-2.5** **2.5-3.0** **3.0-3.5** **>3.5**

What grade do you expect to make in the course? **A** **B** **C** **D** **F**

Major:

Part 2: Background

Where did you first learn to program in Java / C++? (please circle one)

BeforeHighSchool **HighSchool** **College** **Other:**

Do you have any experience programming outside a classroom environment? If so, please explain.

Part 3: Study Reaction

Did you feel that the assigned task was difficult please circle the level of difficulty)

?

What level of difficulty would you rank it? (**Easy** **Difficult** **Hard** **Impossible**
Other:_______________________

What was the most difficult part of this assigned task? Was it the timed aspect of the problem, was the problem too difficult, etc.? Please try to explain in a way that makes the difficulties clear for us.