The Benchmarking Programming Exam and SOA in Introductory Programming Courses

J. Philip East and Andrew Berns
Computer Science Department
University of Northern Iowa
Cedar Falls, IA 50614-0507
east@cs.uni.edu, adberns@cs.uni.edu

Abstract

Assessing student learning or outcomes in higher education instruction is becoming a requirement for all programs and most courses and is increasingly being examined by the Higher Learning Commission as it conducts its institutional accreditation reviews. One way to assess outcomes is through the administration and scoring of a well-designed exam. For introductory programming courses, however, few publicly-available such exams exist. While seemingly developed for teachers of computer science, the Praxis exam in computer science is a possible method for assessing student programming capability. However, it is a commercial product, covers more than introductory programming, and cannot be easily incorporated into regular course assessment, e.g. final exams. An alternative would be to include benchmarking questions produced by Simon et al. These questions are easy to include in a final exam, are relatively easy to grade, and test many of the basic coding elements of programming. By themselves, however, these benchmarking questions do not measure the higher level of comprehension we wish introductory students to have.

In this paper, we propose combining the benchmarking questions of Simon et al. with another measure of programming capability to produce a very effective student outcomes assessment (SOA) instrument. With some care, this combination can provide a measure appropriate for assessing student capability in the current course and for comparing instruction from: semester to semester, instructor to instructor, or course to course.

1 Background

Student outcomes assessment (SOA) has been a topic of discussion and concern for about 20 years. Having an SOA plan in effect is now required for accreditation by the Higher Learning Commission [4]. The goal of this requirement is for there to be an assessment that allows for data-driven decisions about possible course revisions which improve student learning in each course and program. The Commission's website [4] states:

> 4.B. The institution demonstrates a commitment to educational achievement and improvement through ongoing assessment of student learning.
> 1. The institution has clearly stated goals for student learning and effective processes for assessment of student learning and achievement of learning goals.
> 2. The institution assesses achievement of the learning outcomes that it claims for its curricular and co-curricular programs.
> 3. The institution uses the information gained from assessment to improve student learning.
> 4. The institution's processes and methodologies to assess student learning reflect good practice, including the substantial participation of faculty and other instructional staff members.

Clearly, it is intended that we develop sets of goals/outcomes for our courses, devise and administer assessments of student performance relative to these goals, and use data from the assessments to improve instruction. A key to success in this endeavor is the assessment instrument.

There are several commercial and private exams for computer science. The Computer Science Field Exam from ETS [3] is an instrument that can be used to evaluate a CS program or areas within a program. While some areas of the exam may map to specific upper division courses, it does not contain questions which are appropriate for students in introductory programming courses. Designed for computer science teachers, the Praxis exam in computer science [2] more directly addresses introductory programming topics, but it also addresses additional topics from data structures and computer organization that do not relate to introductory programming. Tew and Guzdial developed and validated the Foundational CS1 (FCS1) assessment [8] which considered concepts specific to CS 1. In particular, Tew and Guzdial identified the following concepts for the FCS1 ([7], p.100):

> The final list of constructs which serve as the basis of the test specification are as follows:
> - Fundamentals (variables, assignment, etc.)
> - Logical Operators
> - Selection Statement (if/else)
> - Definite Loops (for)
> - Indefinite Loops (while)
> - Arrays
> - Function/method parameters

- Function/method return values
- Recursion
- Object-oriented Basics (class definition, method calls)

Unfortunately, the instrument from Tew and Guzdial is not publicly available.

Currently, there appear to be no general assessment instruments designed explicitly for introductory programming that can serve to provide data useful in analyzing student learning for the purpose of SOA to improve instruction in introductory programming classes.

2 A Suggested Process/Mechanism

While we have not encountered a fully developed and freely available SOA instrument/process, we have found other work that can form part of an SOA instrument. Simon et al. [5] devised a "benchmarking" exam specifically to be used in introductory programming courses. It can be used to assess basic concepts and skills. We believe it is a good start toward a quality SOA process for introductory programming.

2.1 The Benchmarking Items

The benchmarking assessment tool of Simon et al. [5] was designed and developed by faculty from a number of disparate courses that were offered in different programming languages at seven different institutions in five different countries. The instrument consists of ten items addressing common programming concepts that can be included in the final exam for the course, forming an estimated fourth of the exam. Thus, individual faculty may include assessment items beyond the ten benchmark questions for use in assigning student grades. Additionally, the benchmarking items can be graded in any manner individual instructors wish for the purpose of assigning grades. For comparing results between courses and schools, however, the benchmarking items need to be treated identically. Thus, in addition to developing items for the instrument, the authors also produced detailed marking guidelines for them. If/when one wishes to compare outcomes, the authors' marking guidelines would be followed. Guidelines were developed for several programming languages: Java, Python, C, and Visual Basic.[1]

[1] We received the marking guidelines by requesting them from the first author, Simon, and would be happy to pass them along.

In the marking guidelines document for Visual Basic the items are described in several ways ([6], p.1):

> Here is the breakdown of question forms:
> - multiple choice (4 questions)
> - short answer (3 questions)
> - written code (3 questions)
>
> The questions are loosely grouped into the following topic areas:
> - expressions (1 question)
> - assignment and sequence (1 question)
> - swapping and shifting (1 question)
> - selection (2 questions)
> - iteration and arrays (5 questions)
>
> The questions cover a variety of skills:
> - tracing code (5 questions)
> - explaining code (2 questions)
> - writing code (2 questions)
> - modifying code (1 question)

The benchmarking questions can be further understood by considering where they fit in a taxonomy considering both the *topic* and the *cognitive level* of the question. We analyzed each of the ten benchmarking questions to determine which of the constructs presented by Tew and Guzdial [7] were required to answer each question. The mapping from benchmark questions to constructs is given in Table 1.

Question Number	Construct
1	Fundamentals, Logical Operators
2	Fundamentals
3	Fundamentals
4	Fundamentals, Logical Operators, Selection
5	Fundamentals, Logical Operators, Selection
6	Fundamentals, Logical Operators, Selection, Indefinite Loops
7	Fundamentals, Selection Statement, Definite Loops, Arrays
8	Fundamentals, Definite Loops
9	Fundamentals, Definite Loops, Arrays
10	Fundamentals, Definite Loops, Arrays, Function parameters, Function return values

Table 1: Classification of Benchmarking Questions

We also considered the level of knowledge that was required to answer each question. For this, we used the levels from the cognitive domain in Bloom's taxonomy [1]. As one might expect, all of the ten questions were at a lower level in Bloom's taxonomy. In particular, we classified each question as being at the c*omprehension* level. While it is true that some questions required students to create code, we note Bloom's distinction between *comprehension* and *application*: "A demonstration of 'comprehension' shows that the student can use the abstraction when its use is specified. A demonstration of 'application' shows that he will use it correctly, given an appropriate situation in which no mode of solution is specified." (p. 120).

The benchmarking items address all the constructs suggested by Tew and Guzdial [7] *except* recursion and object-oriented elements programming. We therefore feel comfortable including them in an SOA assessment of our introductory programming courses. They offer a mechanism whereby student success with specific basic skills can be assessed. If, for example, students performed below expectations or desired level on the first item that examines variables, assignment, and sequence the instructor could review course instruction for possible revision to address some particular issue associated with that item. The same appears, to us, to be true of the other items. Each provides the possibility for feedback in the instructional process at a much finer level of granularity than most final (and many mid-term) exams provide. We feel confident that examining the results will provide useful feedback for us.[2]

Our plan for using the benchmark items is to include them in our final exam(s) and check them in two passes. The first pass will grade the final exam for determining student grades. At the current time this will essentially follow past practice in grading final exams. Once grades have been determined and reported a second pass in grading will be used for SOA purposes. In this second analysis we intend to follow the marking guidelines of Simon, et al. Initially, we will use two metrics for measuring success. One measure will be a comparison of our students' results with the results produced from other institutions using the benchmark, the scores of which were included in the original benchmarking paper [5]. We hope to achieve scores similar to other institutions. A second measure will be our reaction to the results. In particular, we wish to determine if student performance, *as a class*, is below our expectations (whatever they were). In both cases, the action we take will be to examine our instruction for possible revision/improvement.

As we become more familiar with checking the benchmarking items, we may well start using a single pass similar to that suggested by Simon, et al. [5]. This lessens the work required for the SOA, leaving only the extra work of tabulating and analyzing the results.

While the benchmarking items are useful, probably necessary, for providing feedback for instructional improvement they are not sufficient, i.e., they do not provide insight into the

[2] We have included the items in some of our courses but have not yet done any analysis for the purpose of student outcomes assessment.

more general aspects of programming like problem understanding, solution development, and code production, nor do they measure skills at the *application* level of Bloom's taxonomy. Our view of programming pedagogy is that students' conceptual understanding and skill as exemplified by these benchmarking items is a prerequisite to more general programming capability. Any SOA instrument would need to include assessments of such student capabilities. The discussion below addresses that additional piece of the SOA puzzle.

2.2 And a Programming Problem

We believe that if you want students to learn to program you must provide them with minimally-specified problems to program. A reasonable conclusion based on that belief is that to assess programming ability, you must have students attempt to produce a program for a minimally-specified problem. Clearly, learning to program will require repeated practice of some sort. The assessment of learning, however, can likely be done with a single problem so long as its elements have been fully exposed by the repeated practice and it addresses the bulk (if not all) of the elements of programming.

2.2.1 Elements of Programming

As noted above, Tew and Guzdial [7] provide one perspective as to what constitutes the basics of programming. We share another perspective with our students, telling them that programming consists of the following:

- *Data* and *actions*
  Data is "the" basic of computing: representing a problem as data for the computer to operate on is key to programming. Once the problem has been represented as numbers and/or characters (or collections of them), the relationships between data elements and their manipulation become the focus of programming.

  While computer scientists have internalized manipulation-related elements of programming, novice programmers must explicitly learn them. The actions on/with data constitute instructions for computers to carry out: input, manipulation/assignment, and output. There are alternatives for each but the most involved is manipulations/assignment, which includes notions of data types and primitive operations and functions.

  Students must be able to represent the problem appropriately and determine the manipulations of the representation that are required to solve the problem.

- *Organizing* the actions via
  - Sequence
    The first and simplest way to organize actions is to sequence them or put them in the appropriate order. Instructionally, we often do not even mention this manner of organizing actions until other ways of doing so are introduced. *Sequence* is always critical and becomes more complex as other organizing schemes are introduced.

  - Selection
    An additional mechanism for organizing actions for the computer is to select from two actions or sets of actions. Sometimes the choice may be to select a set of actions or to do nothing. *Sequence* continues to be used and becomes more complex as actions may occur before, after, or within either choice of the selection construct.

  - Repetition
    Another mechanism for organizing actions for the computer is to allow some actions to be repeated. Repetition (in a realistic situation) will often involve *selection* and always require *sequence* as actions must occur in the appropriate order, whether this be repeated actions or actions occurring before or after the repeated actions.

  - Modularization
    Modularization is the creation of new higher-level or programmer-defined actions. The modules or new instructions created can involve any of the other organizing mechanisms. It is used as to reduce the amount information a programmer needs to keep in mind or to reduce the use of duplicated code.

In short, programming requires "merely" learning the basic actions the computer can carry out and organizing them appropriately to solve the problem at hand. In terms of Bloom's taxonomy, programming requires *comprehension* of the basic actions of a computer and alternatives for organizing them *and* then the *application* (of that knowledge/comprehension) to the programming problem at hand.

It seems pedantic (or worse) to include the above material in this paper. The goal of our doing so is to illustrate our thinking about what needs to be included when assessing student learning of programming. This is what we want students to be able to do. So, based on this mindset, what would we include in our question meant to assess student programming ability?

2.2.2 Problem Characteristics

Our goal with respect to SOA is actually to produce a set of moderately-sized problems from which we can randomly select a single problem to use on a final exam. The problems need to 1) be "equivalent", 2) reflect the content of the course, and 3) exercise most if not all of the fundamental elements of programming. As we contemplated the problems, we worked to characterize what the problems would be and what they should include. Eventually we concluded that the problems would have the following characteristics:

- Be non-trivial, but not complex, and be minimally-specified so as to require some interpretation and consideration of data to represent the problem—application of programming knowledge.
- Require sequence, selection, and repetition, but not modularization as problem size/complexity would typically not reasonably require modularization. The problem's solution should use repetition twice, either as repetition following repetition or a simple nesting of repetition within repetition.
- Involve a collection (accessed via an index) of either string or numeric data and the use of indexing for actions beyond simple traversal of all elements in the collection. This increases the likelihood of non-trivial problems and adds a bit of complexity to problem representation.
- Include selection within at least one instance of the repetition to increase the complexity of the algorithm needed for the solution.
- Include either file input or output to aid in devising realistic problems that require collections, repetition, and selection.

We sought to develop a set of at least five problems that could be shared with students as examples of the kinds of problems for which they should be able to write programs for at the end of the course.

One of us (Philip) has used problems like this in the past as part of the final exam. We intended to use this list of problems as part of this paper. However, when we analyzed the problems we discovered that they were not comparable problems. We therefore generated a new list of problems which appears in Appendix A.

2.3 Administering the Final Exam (& SOS Assessment)

For us, the final exam is the same as the SOA assessment. We desire particular outcomes for the course and both the exam (for "grading" students) and the SOA (for "grading" the instruction) can reasonably be the same. The students are informed of the nature of the final in a document describing, in general, the benchmarking portion of the exam and listing a larger number of sample programming problems. They are told that the exam will consist of the 10 items assessing basic skills and concepts and that there will be a

programming problem similar to those shown. Typically, the programming problem will be taken from the list provided the students. (If the students decide to crib for the exam by programming all the problems, so much the better.)

The goal of the exam is to check to see if students can demonstrate the desired performance outcomes. Thus, the exam is closed-book and closed-notes. We check the program code produced by the students for conceptual and semantic understanding, not syntactic correctness. As with the benchmarking items, marking the program for student grading need not be the same as marking for SOA purposes. We tend to be more holistic when examining student responses for purposes of assigning grades. A detailed analysis of the responses, however, is called for when assessing for SOA purposes.

2.4 An SOA Analysis Rubric

A rubric is required to effectively and consistently determine which elements of programming students can appropriately apply in solving the given problem. Below we provide a high-level rubric which identifies the elements of a generic solution to our problems that can be considered to measure student learning. This rubric is intentionally general so as to apply to all of our problems.

As a note, we have not yet administered an exam with our problems and obviously have not had the chance to use our rubric. We welcome any feedback regarding the rubric.

A solution should include the following elements, grouped by "type":

- File-related
    - File is properly opened and closed
    - Code for reading from and writing to the file is located appropriately (e.g. within repetition)
- Selection
    - Correct Boolean expression used
    - Appropriate conditional structure selected (if-then vs if-then-else vs …)
    - Appropriate action(s) taken for each outcome
    - Actions are not unnecessarily repeated
- Repetition
    - Correct continuation/halting condition
    - Actions repeated where appropriate
    - No unnecessary actions repeated
- Sequencing actions
    - Appropriate actions taken before and after repetition
    - Correct sequence of actions performed within repetition:
        - Before, inside all outcomes, and after selection
- Correct non-trivial use of an index into the collection

3 Summary

We have discussed and described combining a comprehension-based assessment of introductory programming with an application-level assessment to produce an SOA (student outcomes assessment) instrument. In so doing we noted the current lack of an appropriate instrument for introductory programming courses and the existence of the benchmarking items suggested by Simon et al. While useful, the benchmarking exam cannot be the sole assessment of programming skill as it tests only comprehension and not the application of programming knowledge to produce programs for minimally-specified problems (a key goal in most programming courses). We suggest adding a single programming problem to the benchmarking items to produce an instrument that can serve as both a final exam and an SOA instrument, albeit with perhaps separate making schemes for each purpose. Characteristics of the problem are indicated along with a suggested marking scheme. Sample problems are also provided.

The goal of SOA is to improve instruction. An appropriate SOA instrument can be used in a variety of scenarios. Its first use would be to determine or set performance levels considered as constituting success in the course. After establishing baseline performance and student population characteristics, the SOA results can be used to judge both specific and overall indications of instructional success. Areas of difficulty can be identified and revisions to instruction made and ultimately tested. Those interested in testing alternative pedagogical approaches can use the same instrument to compare current student outcomes with the historical/baseline values (updating the baseline at the same time). Schools often have different programming courses. The proposed instrument could be used to compare outcomes in those courses. There is already a baseline for the benchmarking portion of the instruction across languages, institutions, and countries with which one can compare outcomes. When/if the programming problems become more widely used a more comprehensive assessment of student outcomes can be made.

The goal of this paper has been to suggest one possibility for gathering quality SOA data for introductory programming courses/experiences. Using an SOA instrument will allow for data-based decisions about programming instruction in addition to the instructors' sense of the success of the course. We do not wish to belittle instructor intuition, we simply suggest that we not rely on it alone when making decisions about teaching and learning.

4 References

[1] Bloom, B.S., *A Taxonomy of Educational Objectives: Handbook I: Cognitive Domain*. Longmans, Green and Company, N.Y. 1956.

[2] ETS (Educational Testing Service). The Praxis Study Companion—Computer Science (5651). Retrieved from https://www.ets.org/praxis/prepare/materials/5651

[3] ETS (Educational Testing Service). The Major Field Test for Computer Science. Retrieved from https://www.ets.org/mft/about/content/computer_science

[4] HLC (Higher Learning Commission). Criteria for Accreditation. Retrieved from https://www.hlcommission.org/Policies/criteria-and-core-components.html

[5] Simon, Sheard, J., D'Souza, D, Klemperer, P., Porter, L. Sorva, J., Stegeman, M. & Zingaro, D. Benchmarking Introductory Programming Exams: Some Preliminary Results. *ICER'16*, September 8-12, 2016, Melbourne, Vic, Australia.

[6] Simon, Sheard, J., D'Souza, D, Klemperer, P., Porter, L. Sorva, J., Stegeman, M. & Zingaro, D. Benchmarking programming exam questions (Visual Basic version). Received by request from an author (Simon).

[7] Tew, A.E. & Guzdial, M. Developing a Validated Assessment of Fundamental CS1 Concepts. *SIGCSE'10,* March 10-13, 2010, Milwaukee, WI, USA

[8] Tew, A.E. & Guzdial, M. The FCS1: A Language Independent Assessment of CS1 Knowledge. *SIGCSE'11*, March 9-12, 2011, Dallas, TX, USA

Appendix A
Sample Problems for Final Exam & SOA

Process the scores recorded in a file (`scores.txt`) to calculate their mean and report: 1) the mean, 2) the number of scores above mean, 3) the number of scores below mean, and 4) the median. Process the file one time only. (Note: the mean is the sum of the scores divided by the count of scores and the median is the middle score when the count is odd and the mean of the two middle scores when the count is even.)

Process lines in a file of text to report the message hidden inside by a cipher. The message consists of all the characters that appear after a space in the original text (including spaces that occur after a space).

Process a file of words and word counts that are sorted by word count values. The goal is to produce a file of non-stop words (words to be included) in a word cloud. The top and bottom 10% of the words **and any words with the same counts as those** are stop words. Process the file one time only.

Provide drill and practice on states and capitals until the user responds with "stop". The state and capital values are in a file (`states.txt`) as comma separated values with one state and capital per line. Randomly select a state to present and check the user's response for correctness. When halting, report the percentage of state and capital pairs the user correctly identified.

Process a list of words to count occurrences of various lengths of words. Report the non-zero occurrences in increasing order of word length. Assume a maximum word length of 50 characters.