

## Position Paper for FYI 2000

**Mike Clancy (University of California, Berkeley; [clancy@cs.berkeley.edu](mailto:clancy@cs.berkeley.edu))**

Berkeley has four lower-division courses intended for computer science majors. The first, CS 61A, is organized around the text *Structure and Interpretation of Computer Programs* [1]; the second, CS 61B, is a Java-based data structures and intermediate programming course; the third and fourth (outside the scope of this position paper) are courses in machine structures and discrete mathematics. In addition, we run a Scheme-based introductory course, CS 3, for non-majors in which we focus on functional programming. (The document

<http://buffy.EECS.Berkeley.EDU/~clancy/evolution.html> describes the evolution of these courses.)

Our lower-division courses are intended to provide students with programming skills (program analysis and synthesis; language, tool, and library use), knowledge of techniques and concepts to be reinforced and extended in subsequent courses, and an overview of computer science sufficient to allow a realistic decision about majoring in the field. The first-year courses address these goals as follows:

- CS 61A gives students practice with a variety of programming paradigms (functional, object-oriented, and rule-based) in an attempt to broaden their programming repertoire. Our treatment here is based on the adage that everything looks like a nail when one's only tool is a hammer. The course also covers the techniques used to implement the primitive operations for those paradigms, thus preparing students for more intensive study of data structures, compilers and interpreters, and advanced techniques treated in depth in a variety of upper-division courses.
- CS 61B introduces students to Java, building on their experience with object-oriented programming in CS 61A. It then covers the data structures topics of the traditional "CS 2": arrays, linked lists and trees (previously covered in CS 61A), heaps, hash tables, and sorting methods. Though students do relatively little algorithm analysis in this course, they at least are exposed to the activity and its place in the broader context of computer science. CS 61B is also the first place in the curriculum where students design and develop relatively large programming projects (1000-1500 lines of code) from scratch, and includes coverage of tools and techniques that support productive programming. Techniques and concepts covered in CS 61B reappear throughout the upper-division courses.

All the lower-division programming courses involve the use of "closed labs". In CS 61AB, labs provide closely supervised practice with language constructs and low-level techniques, as well as opportunities to experiment with algorithm and data structuring alternatives and to display progress and get feedback from t.a.s on projects.

CS 3 has benefited from NSF-supported exploration of aspects of learning Lisp [5, 7, 9] and development of case studies that model the process of programming [3, 4, 8]. We are gradually applying knowledge gained in CS 3 to the other first-year courses.

We are generally satisfied with the organization and content of these courses. Several challenges, however, remain:

- Many students still find it traumatic to move from Scheme in CS 61A to Java in CS 61B, and

from Java to C in later courses. Pedagogical research is needed in a number of areas: What misconceptions do students encounter when introduced to a syntax-rich language, and how can we address them? How can we encourage transfer between languages? What support can we provide to students using relatively low-level programming constructs and techniques? (These are instances of a general problem, namely that the technology we use to teach programming is so far ahead of the pedagogy.)

- Few of our students can design, manage, and analyze complex code. (Some of my colleagues, facing the same problem, argue for postponing instruction about design beyond the first year. We, however, are more optimistic.) How can we help? In particular, what techniques and skills are amenable to modeling in case studies, and what other support can we provide students in structured activities and staff consultation?
- We're still learning about object-oriented design ourselves. It's much harder than procedural design; one must worry not only about how to decompose a program into pieces, but about how those pieces interact. Software design patterns [6] might provide the basis of effective teaching of object-oriented design skills, but they require substantial pedagogical groundwork [2].
- The Java 2 libraries include classes for almost all the standard data structures, as well as facilities for implementing graphical user interfaces. How can we help students effectively learn the fundamental techniques of linking, unlinking, insertion, and deletion without reinventing the builtin wheels? To what extent does the motivational benefit of creating and using a modern interface outweigh the additional complexity involved in using GUI classes? How can these facilities be used to provide better tools for instrumenting and visualizing programs?

I'm looking forward to gathering ideas that address these concerns at the FYI workshop.

#### References:

1. Harold Abelson and Gerald Jay Sussman, *The Structure and Interpretation of Computer Programs* (second edition), MIT Press, 1996.
2. Michael J. Clancy and Marcia C. Linn, "Patterns and Pedagogy", proceedings of the 30th SIGCSE Technical Symposium on Computer Science Education, New Orleans, Louisiana, March, 1999; published as *SIGCSE Bulletin*, volume 31, number 1, March 1999.
3. Michael J. Clancy and Marcia C. Linn, *Designing Pascal Solutions: Case Studies with Data Structures*, W.H. Freeman and Company, 1996.
4. Michael J. Clancy and Marcia C. Linn, "Case Studies in the Classroom", proceedings of the 23rd SIGCSE Technical Symposium on Computer Science Education, Kansas City, Missouri, March, 1992; published as *SIGCSE Bulletin*, volume 24, number 1, March 1992.
5. Elizabeth A. Davis *et al.*, "Mind Your P's and Q's: Using Parentheses and Quotes in Lisp", *Proceedings of the Fifth Workshop on Empirical Studies of Programmers*, Curtis R. Cook *et al.* (editors), Ablex Publishing, 1993.
6. Erich Gamma *et al.*, *Design Patterns*, Addison-Wesley, 1995.
7. Christopher M. Hoadley *et al.*, "When, Why, and How Do Novice Programmers Reuse Code?", *Proceedings of the Sixth Workshop on Empirical Studies of Programmers*, Wayne D. Gray and Deborah A. Boehm-Davis (editors), Ablex Publishing, 1996.
8. Marcia C. Linn and Michael J. Clancy, "The Case for Case Studies of Programming Problems", *Communications of the ACM*, volume 35, number 3, pages 121-132, March 1992.