

Teaching Philosophy

Anne Mulhern

November 13, 2008

Computer science is not a spectator sport. This has been said frequently about mathematics, but is no less true of computer science. Fortunately, computer science has provided us with computers; we can interact with computers and do, not just observe.

At bottom, computer science is a wonderful, free-wheeling mathematical subspecialty with a bit of engineering thrown in. An introductory computer architecture course is just a study in Boolean algebra applied in ingenious and every more intricate ways. Computer science theory is mathematics, through and through. Machine learning is a fascinating application of probability, statistics, and a number of other mathematical disciplines. The study of programming languages is also a study in logic; the Curry-Howard isomorphism shows how logics on the one hand and programming languages on the other can be identified.

However, computer science often seems to have very little to do with mathematics, especially when it comes to introductory courses. This is due to the wonderful freedom of computer science and its hands-on nature; in computing it is really easy to build things that don't quite work. A frequent example is that of programming languages that were designed to be type-sound, but turned out not to be, e.g., Java or Eiffel, or of programming languages, like C, with a static type system that were designed before the notion of type-soundness was clearly understood. This freedom is a strength of computer science as well as a weakness; learning from mistakes is a wonderful way to learn.

The key to teaching computer science is to teach the mathematics while simultaneously taking advantage of all the opportunities computers allow for experimentation. For example, recursion in computer science and induction in logic are two sides of the same coin. There are certain conditions that must hold true for a proof by induction to be valid. If the same conditions do not hold true for a recursive function, there is some input on which the function will continue executing forever. Since computers are finite an infinite recursion will cause the computer to hang or crash; for most students either of these program behaviors is a powerful kind of feedback.

If computer science is hands-on mathematics with almost immediate feedback, then what should really be taught in a computer science course? I believe that the following principles should guide any curriculum.

- Principles are more important than details.

- Active experimentation is more valuable than passive absorption.
- Mistakes, crashes, incorrect results and so forth, are not just things to be fixed and swept under the rug, they are things to be understood and learned from.

My teaching experience includes introductory programming courses using Java at the University of Wisconsin and at Sarah Lawrence College, a fundamentals of computer science course for gifted middle and high-school students, and work as a teaching assistant for various courses in the Mathematics and Computer Science departments at Wisconsin. I currently teach a seminar on automated theorem proving using the Coq proof assistant.

My experience in teaching introductory programming courses using Java has helped to intensify my belief that principles are vastly more important than programming language details. Too often, students struggle with the syntax of Java, losing themselves in the details of class definitions and matching braces. Moreover, they often think that their smartest questions are stupid. For example, students have to be reassured that the question, "How does the computer take my source code, which is just text, and actually make it do something?" is a profound and vital computer science issue and not a shameful admission of ignorance about something that is really just plain obvious. Clearly, something is going wrong here.

How can things be improved? Techniques and tools have been introduced to make programming more accessible to novices. For example, the TeachScheme! project follows a methodology of introducing progressively more complicated subsets of language to students and this idea has been extended to Java by the TeachScheme! Reach Java project. Programming environments designed specifically for students, e.g., BlueJ, also make it easier for beginning students to write programs. The Python language seems particularly accessible to beginning students; some believe that this is due to its use of indentation to indicate nesting, others point to the fact that one can interact with it through the top-level, thereby getting immediate feedback without having to go through the write-compile-execute cycle, still others believe that this is because it is dynamically checked, and students can deal with runtime errors as they occur rather than struggle to satisfy the type-checker, whose purpose and error messages are generally obscure to many novice and even experienced programmers.

Addressing the syntax barrier is important and all these approaches have their merits. However, it is just as important to discuss principles early. Students should be assisted to develop the ability to reason about their programs as soon as they start running them. The TeachScheme! Reach Java project takes a sound approach by introducing mutation, i.e., variable assignment quite late, rather than immediately, and motivating it carefully. Recursion is so fundamental that it should be discussed explicitly, before looping structures. Computer scientists know that loops and mutation are in general very hard to reason about; novice programmers interpret their own confusion as stupidity or lack of ability.

I've used teaching introductory programming as an example, but these ideas apply just as well to other subjects. I can barely remember first learning to write programs but I can still remember my introductory compiler class; I implemented a working type-checker without having any clear notion what a type-checker was for. This contradicts my first principle. My second principle was also contradicted; I followed a recipe for constructing a type-checker but I never had time or energy to consider what would be the effect on the programs compiled by my compiler if the type-checker didn't work. My third principle was contradicted as well; never having programmed in a dynamically checked language, I had no experience with the kinds of errors the type-checker was intended to prevent.

In my seminar on automated theorem proving using the Coq theorem prover I make every effort to follow these principles. The theorem prover is powerful and comes equipped with many powerful tactics; I avoid constructing my course around the tactics and instead discuss the underlying principles at work. My students must solve the goals themselves; I don't push them through a series of cookbook solutions. Theorem proving is a demanding discipline and it is easy to go wrong by changing a provable goal to an unprovable one by a poor choice of proof strategy; when my students make these poor choices I help them to reflect on their mistakes.

Last, but not least, respect for one's students is fundamental to good teaching. It is certain that their backgrounds and perspectives are different; they may bring valuable insights to the class. If they are really struggling, as some do with what can be a very difficult subject, this should not be viewed as an indictment of their intelligence or character. They may simply be unprepared or their strengths may lie in other areas. Guy Steele, one of the principle designers of the Scheme and Java programming languages, once described to me why he chose to drop a category theory class. He has gone on to an extremely distinguished computer science career. It is far better to take a wider view than to consider a student's current performance in one's class as the measure of that student's worth.