From *How I Teach*, V. III: Essays by winners of the Goergen teaching award, Univ. of Rochester, 2005.

Teaching Intellectual Connections

Michael L. Scott
Department of Computer Science

As an academic I have written thousands of pages on my research, but I have never before been asked—and have certainly never presumed—to write about my teaching. It's a humbling exercise.

My mother has an M.A. in Teaching from Harvard. My father has a professional degree in Religious Education. My advisor in graduate school, Raphael Finkel, completed his M.A.T. at Chicago before pursuing a scientific doctorate. And here at Rochester we have a graduate school of education and human development, full of bona fide experts on teaching. But me? I've never taken a single course on the subject. My credentials are just on-the-job experience: 20 years as a student and 20 more as a teacher.

My position, of course, is not unusual. Unlike our colleagues in K–12 education, most college professors have no formal training as teachers. Our student careers were entirely devoted to reaching and extending the intellectual boundaries of our fields. To profess, after all, is to make an open declaration of one's knowledge and belief, and this is what we do. The quality of our knowledge and belief is generally very high, but that of our declarations is sometimes hit-or-miss.

Like most of my peers in academia, I am here because I love what I do. I am devoted to my field and to sharing it with students. Computer science is unique in its fusion of abstract structure with tangible working artifacts. It has sometimes been described as the discipline for people who love puzzles: logical puzzles, crossword puzzles, jigsaw puzzles, mechanical puzzles, or games like chess and go. Unlike the natural sciences, computer science is not constrained by physical reality. Unlike engi-

neering, it is inherently precise. And unlike mathematics, which describes so many systems, it embodies those systems directly.

If I had chosen to study religion or history or law (all of which I considered at one time or another), I would have missed some wonderful opportunities, but I would probably still have ended up a teacher. Part of the motivation is undoubtedly innate, but much of it comes from having had wonderful role models. Wayne Wolfgram, who taught my high school trigonometry and discrete math courses, and coached the school's math team, made mathematics not only interesting but fun. Tom Keough poured his heart into high school English, and directed me in four school plays. Raphael, my graduate advisor, and his colleague Marvin Solomon exemplified the fusion of teaching and research, and Sam Bent, from whom I took about six different graduate courses, conveyed more enthusiasm for his subject matter than six typical teachers combined. If not formally then at least by example, they taught me how to teach. I am deeply indebted to them all and to others too numerous to mention.

My job as a teacher, as I see it, is to help students experience the excitement and enthusiasm that comes from deep understanding of a rewarding subject. To do that job well I have to know and love the material, I have to empathize with students, and I have to apply good techniques. As an experienced if amateur teacher and a sometime mentor to younger colleagues, I often ask myself: How much of good teaching is talent, and how much can be learned? I can learn to be an expert in my subject, but I can't convey enthusiasm unless I feel it myself. I can learn to interact with students—to ask them about their lives and about their experience in my course—but I also think there's something innate about being able to model other people's thoughts: to imagine myself in their shoes and figure out what they need to hear in order to understand.

Perhaps the biggest learnable part of teaching is concrete classroom skills. Like most experienced teachers, I have a sense of what works and what doesn't work for me, but I'm sure there's much I could learn from the folks who study teaching for a living. In particular, I'm conscious of how little I know about different styles of learning: about how the mind assimilates data, how that varies from person to person, and how I might encourage it with better concrete skills. At the same time, I'm convinced that

lecturing, at least, involves a significant amount of performance—of theater, if you will—and while much of that can be learned, a good part, I think, is innate.

Over the course of 20 years I've learned which parts of the job I do well. I have a knack for explaining things, both verbally and in writing. I'm also a bit of a ham, at least in structured settings, which serves me well in lecture. I enjoy explaining things, and I never feel I truly understand them until I can explain them. As a graduate advisor I emphasize the background, the context, and the implications of research as much as the technical details. I tell my students that even the most brilliant work will never change the world unless they can explain it.

I also know what I don't do well. I'm not by nature a social person. (I've never known what to do at parties.) Outside the structure of the classroom I struggle to connect with students. I find it hard to learn about their interests, their families, their hopes, and their fears. Only a few take advantage of my office hours. I think I'm seen as caring, but not particularly approachable. I'm better at connecting than I once was, but it's something I really have to work at. I stand in awe of other teachers for whom such personal connections are easy. Within my own research group, I particularly admire the success of my two youngest colleagues at promoting undergraduate research. Undergraduates need more personal attention, direction, and support than graduate students do, and I'm simply not as good as they are at providing it.

So what about the learnable part? What is it that works for me? The answer is two-fold. First, I have found that many of the most interesting ideas, the deepest insights, and the most compelling educational lessons arise when we find connections among fields that were previously viewed as independent. Second, I have learned three crucial practical lessons: be prepared, be interesting, be humble.

Consider the issue of connections. For years I have wondered, when filling out my income taxes, whether to list my profession as College Professor or Computer Scientist. One of the joys of working at the University of Rochester is that I can really and truly be both. Another joy is that I can afford, both in teaching and research, to be a generalist.

Within computer science, my research spans the boundaries of operating systems, programming languages, and computer hard-

ware design. In a larger context, members of my department collaborate actively with researchers in some 20 other disciplines. These connections, both within and among departments, are easier to find at a smaller institution, because everyone rubs elbows with people in other fields. The resulting cross-fertilization is one of the principal attractions of working at a place like Rochester.

During my first few years as an assistant professor, I twice taught our course in compiler design. A compiler is the tool that translates from a programming language like C or Java into the binary control language of some particular model of computer—the Intel Pentium, for example. Then in 1989, with the departure of one of my colleagues, I inherited the course in comparative programming languages. At most schools these courses are taught independently, with neither a prerequisite for the other. But many of the most important decisions in programming language design are driven by concerns over how easy or difficult it will be to compile a given language feature, and many of the most important issues in compilation are driven by the details of language design.

With limited teaching staff, we opted to drop the compiler course and merged some of its content into the languages course. I had always believed that it made sense to cover the subjects together, and this conviction grew over time. Increasingly frustrated with the traditional comparative language textbooks, I set out in 1995 to write a text that would emphasize the interplay of language design and language implementation. The book was completed in the summer of 1999 and was published in October of that year (*Programming Language Pragmatics,* Morgan Kaufmann Publishers, 2000). Conversations with colleagues across the country and around the world suggest that the integrated approach is increasingly appealing, and indeed the book has been highly successful. The second edition will be released in October 2005.

On a much less ambitious scale, I championed a redesign of our computer organization course in the spring of 2002, using the innovative text of Bryant and O'Hallaron. Where the course had previously focused on computer hardware design, it now emphasizes the impact of the hardware on operating systems, compilers, and application programs.

In a field that changes as rapidly as computer science, it's important to stay fresh. I can never teach a course from the same

set of notes in consecutive years. They always need updating. This leads to the first of my practical lessons: Be prepared. Like most teachers, I use detailed lecture notes, and I can always tell when I haven't put as much time into them as they deserve. In the languages course, for example, a standard pitfall is to illustrate a topic with a toy program: an artificial example that captures some language feature but doesn't do anything interesting. If a feature is worth discussing, then it matters in some real program. I don't necessarily have to find such a program out in the business world somewhere, but I owe it to my students to craft an example that could plausibly appear in real life.

In many cases, preparation saves time in the long run. A hastily written exam is often harder to grade, and an unclear or inconsistent assignment takes more time to fix or explain than it would have taken to write well in the first place. One of the most valuable roles I have found for TAs is in vetting assignments before I give them to students. It also helps, if I can manage it, to solve each assignment myself, in advance. In systems courses like the ones I teach, it is common to give assignments of the form "write a program to do X." I don't have to write the program myself to know whether a student's version does what it's supposed to do, but if I skip that crucial step I won't have the hands-on experience to offer good advice, and I risk creating an assignment that's too difficult to finish on time.

In many cases, an overly difficult assignment can be simplified by giving students an initial body of code on which to build. For the languages course, I provide a complete compiler for a simple programming language. Students then extend the compiler to translate additional features. In addition to exposing students to issues that they would not reach if writing from scratch, the experience of reading and trying to understand a large existing system provides valuable preparation for real-world software development, which almost always uses large amounts of someone else's code. (Students don't like to read code and often complain about it on their course evaluations, but I figure it's much better to gain some experience now than when facing a real-world deadline, when one's job is on the line.) To ease the task somewhat, I often devote one or more lectures to a tour of the existing code.

Similar preparation is useful in course administration. I provide a wealth of materials on the Web, including course procedures

and expectations, grading standards, my lecture notes, old exams (with answers), a schedule for the semester, details of all assignments, a list of books on reserve at the library, and pointers to resources elsewhere on the Web. I also maintain a "newsgroup" (an online forum), in which students can air questions, concerns, and suggestions. I read the newsgroup every day and post responses whenever appropriate.

I give mostly short essay exams, in which students are encouraged to apply what they have learned, evaluate tradeoffs, and extrapolate to situations that they may not have considered before. I pass out suggested answers to all questions as students leave the room, so they can get an immediate sense of how they've done. Though I rely on TAs to grade homeworks, I grade exams myself. Afterwards, I post new answers on the Web to any questions for which students have made points that did not occur to me initially.

No amount of preparation, unfortunately, will engage students in the classroom. While I have taught a variety of graduate seminars and the occasional undergraduate discussion class, most of my courses are lecture-based, and it is lectures for the most part that shape a course from the student's point of view. This leads to my second lesson: Be interesting. I've already noted that I experience lectures as performance. I get an adrenaline rush out of teaching; it takes half an hour afterward to wind down. I use all the usual tricks: I make a lot of eye contact, I move around a lot, I use humor when I can. But most important I aim for interaction.

If I come into lecture with a fixed agenda and a fixed set of overhead slides, there's little to distinguish my performance from a canned presentation on videotape. The value added for students at a residential college is the opportunity to interact with their instructors, and I strive to help students make the most of that opportunity.

I've found that the very first class period sets the tone for the whole semester. If I don't get students to participate on day one, they probably won't participate at all, and the course ends up dreadfully dull. My first lecture in any class thus begins with a brainstorming exercise, in which I get as many different students as possible to voice a suggestion or opinion. In the programming languages class, I simply ask students to name as many program-

ming languages as they can. We usually come up with 30–50. We then organize these into major linguistic groups and ask: Why are there so many languages? What are the principal ways in which they differ? What makes a language good for a particular purpose? What makes a language commercially successful? Is "good" the same as "commercially successful"? Why or why not? I've found that this sort of icebreaker sets the stage for ongoing interaction. Even in a lecture of 75–100 students, where informal discussions are not practical, I field dozens of questions every lecture.

In recent semesters I have asked my students to let me take pictures on the first day of class, to help me learn their names. It's a somewhat awkward exercise, but I don't have a talent for names, and the camera has finally made it possible for me to learn them all. Students seem to appreciate the effort; they've been good about posing for mug shots.

In general, I try to concentrate in lecture on big-picture issues: putting material in context, providing historical and interdisciplinary perspective, and emphasizing overarching themes. I try not to get bogged down in detail—that's always available from the textbook—but I spend extra time on subjects that students are likely to find particularly subtle or confusing. I watch faces to see who looks confused, and I constantly fish for questions and other feedback. Often I'll stop and ask, "Who thinks they understood that?" "Who needs some more explanation?" It's important to ask both ways and to insist that everyone raise a hand one way or the other. I'm quick to praise good questions and never suggest that there is any other kind. I make clear that a student with a question is doing community service: There are almost always a dozen others who want to know the answer but were too timid to voice their confusion.

Depending on the feedback I receive, I adjust the level, direction, and emphasis of my lectures on the fly. The most interesting classes, both for me and for the students, are often those in which a question takes me off on a tangent—perhaps a historical anecdote, perhaps a discussion of a recent commercial announcement—that I could never anticipate ahead of time. Significantly, this sort of flexibility does not mix well with overhead transparencies or video projection. For this reason I'm a dedicated chalk-and-blackboard lecturer. By limiting myself to paper notes I give myself the freedom to go where the class needs

to go. I also avoid the temptation to cover material more quickly than I can write.

Classroom technology clearly has its place: I'll often create a slide for a particularly complex diagram. But far too often I think slides lead to leaden presentations, and students seem to agree. When I announced at the beginning of a sophomore class a few years ago that I didn't believe in PowerPoint, the students broke into spontaneous applause.

Of course the danger in going off script is that one will make mistakes. This leads to the third of my practical lessons: Be humble. One of the hardest things for me to learn was that it's ok to mess up from time to time. Students are forgiving, particularly if I don't try to hide my mistakes. Often one of the better students will catch errors as I make them, or bail me out if I get lost. In the worst case, I can always move on to the next subject and promise to straighten things out next time. Students also appreciate an honest, "I don't know, but here's how we might find out," in response to an interesting question. The only time I feel I've really failed is if a student comes up to me after class with a correction that I could have had immediately if I hadn't been rushing and thereby discouraging questions.

For many years the programming languages course at Rochester was cross-listed for graduate and undergraduate credit and drew a roughly 50/50 mix of first-year Ph.D. students and self-selected juniors and seniors. In the fall of 2002, a year after I received the Goergen award, we created a separate course for the graduates and made the existing course a requirement for the undergraduate major. In teaching the course that year, I failed to adequately adjust for the change in the level of students. The projects were too hard, the lectures moved too quickly, and the students grew discouraged. By halfway through the semester I knew there was a problem, but I had my entire syllabus mapped out, and I didn't want to change it. At the end of the semester I received—and deserved—the worst evaluations of my teaching career, and I re-learned my third key lesson: Humility requires a willingness not only to stumble at the blackboard but also to change the whole course if it isn't meeting the students' needs.

Before closing I'd like to turn my attention to a pair of issues that concern me. Both relate to teaching, but neither, I think, can be addressed through teaching alone. The first is the need to fos-

ter personal integrity, both in the classroom and in society at large. Surveys suggest that cheating is rampant on college campuses, and that it mirrors the rest of society. Increasingly, Americans seem to see cheating as commonplace and even as acceptable, so long as "everyone else is doing it," and so long as one doesn't get caught. We underpay our taxes; we pad our resumes; we drive 10 mph over the speed limit; we lie in political campaigns. And in college we copy some other student's work.

It's hard for me to tell how much cheating goes on in my classes. I don't see huge amounts, but then I don't look very hard, and I'm shocked that it happens at all. I'm also deeply troubled by the sense that students don't work as hard as they did a generation ago. Maybe I have a warped view of the past. Maybe today's students have too many conflicting commitments. Maybe I'm becoming a curmudgeon. But what am I to make of students who spend $40,000 a year on tuition and board and then don't read the textbook, don't attend class, neglect to turn in their assignments, and complain about their grades? They aren't the norm, thank heavens, but they're a sizable minority, and a growing one, I fear. Fifteen percent of my students are likely to be absent on any given day.

I don't believe that the fear of getting caught is an appropriate deterrent, let alone an effective one. Society works only when people follow an internal code of personal integrity. I can make my standards clear; I can strive to inspire and persuade, but I can't stem the tide of social change, and I fear it may be flowing in a troubling direction.

Finally, I worry about the image of science in society. Paradoxically, while Western civilization is increasingly dependent on technological developments, large segments of the public not only don't understand science, but actively mistrust it. Fewer and fewer students emerging from the public schools have either the inclination or the preparation (particularly in math) to pursue scientific degrees. For at least a generation, America has depended on an influx of students from abroad to fill its graduate programs and its high-tech jobs, and even this source of talent has begun to evaporate, as the booming tech economies of India and China keep more of their students at home, and as post-9/11 immigration restrictions make it harder and less pleasant for foreign nationals to study here.

In computer science, the demographics are particularly bad for women and members of underrepresented minority groups (basically all but Asian Americans). While the overall national percentage of science and engineering bachelor's degrees awarded to women topped 50 percent for the first time in 2000, the percentage in computer science is still less than 30 percent and less than 20 percent at top-ranked schools. For African Americans, the national average in computer science is approximately 10 percent, but only 3 percent at top-ranked schools. These discouraging figures persist despite at least 15 years of intense efforts by colleges, professional organizations, and the National Science Foundation to increase diversity.

If we assume, as seems reasonable, that there is no inherent reason why women and minorities should steer clear of high-tech degrees, lopsided enrollments suggest that we are shortchanging both students and the nation, which is trying to populate the tech economy from a fraction of the total population.

So what are we doing wrong? An increasing body of scholarship makes clear that the problem is deeply rooted in societal norms and perceptions, and that no comprehensive solution will be possible without widespread cultural change. At the same time, there is much that we could be doing individually to change the way we teach—to improve the classroom climate (see for example Seymour and Hewitt's *Talking About Leaving: Why Undergraduates Leave the Sciences,* Westview Press, 1997, or the many resources at cra.org/Activities/craw and at anitaborg.org). Science and engineering departments tend to have an unusually competitive, grin-and-bear-it student culture, which white males seem disproportionately willing to endure, but which serves no one particularly well. Deliberate steps to reduce competition, increase collaboration, minimize drop rates, and affirm and encourage students on a personal level would be good, I believe, for all concerned. This needn't entail any lowering of standards, but it will require more faculty attention, to intervene with struggling students rather than simply dismissing them.

Concurrently, I think that faculty in science and engineering, and in computer science in particular, need to do a much better job of explaining their field to others, both as a near-term recruiting aid and as long-term public relations. We should be able, as no one else, to articulate the excitement we experience as

researchers, and that we already convey in the classroom. We must fight the perception that science is dry, or isolating, or sterile, or amoral. Biologists have won this battle, or very nearly; we must emulate their success. Most of all, I believe, we must make the case that science changes lives, that it can make the world a better place, that it connects with everything else.

And here I come full circle. Computer science is a driving force behind artistic expression, medical imaging, bioinformatics, grass-roots democratic movements, independent news reporting, weather and climate modeling, unmanned space exploration, security and counter-terrorism, evolutionary biology, cosmological and quantum physics, and the quantitative social sciences. These contributions to allied fields are more than applications. They are two-way intellectual connections that will shape the future of computer science, too. In an increasingly interdependent world, similar lists undoubtedly exist for every realm of intellectual endeavor. We owe it to our students, and to society at large, to find the cross-connections, to understand their implications, and to explain them to the best of our ability.