DOI:10.1145/1378704.1378713

Mark Guzdial

## Education

# Paving the Way for Computational Thinking

*Drawing on methods from diverse disciplines—including computer science, education, sociology, and psychology—to improve computing education.*

TEACHING EVERYONE ON campus to program is a noble goal, put forth by Alan Perlis in 1962. Perlis, who was awarded the first ACM A.M. Turing Award, said that everyone should learn to program as part of a liberal education. He argued that programming was an exploration of process, a topic that concerned everyone, and that the automated execution of process by machine was going to change everything. He saw programming as a step toward understanding a "theory of computation," which would lead to students recasting their understanding of a wide variety of topics (such as calculus and economics) in terms of computation.[4]

Today, we know that Perlis was prescient—the automated execution of process *is* changing how professionals of all disciplines think about their work. As Jeanette Wing has pointed out, the metaphors and structures of computing are influencing all areas of science and engineering.[6] Computing professionals and educators have the responsibility to make computation available to thinkers of all disciplines.

Part of that responsibility will be met through formal education. While a professional in another field may be able to use an application with little training, the metaphors and ways of thinking about computing must be explicitly taught. To teach computational thinking to everyone on campus may require different approaches than those we use when we can assume our students want to become computing professionals. Developing approaches that will work for all students will require us to answer difficult questions like what do non-computing students understand about computing, what will they find challenging, what kinds of tools can make computational thinking most easily accessible to them, and how should we organize and structure our classes to make computing accessible to the broad range of students.

Through a few brief examples, I will show in this column how these

ILLUSTRATION BY CHRISTOPHER SILAS NEAL

questions are being addressed by researchers in the field of computing education research. Researchers in computing education draw on both computer science and education—neither field alone is sufficient. While we computer scientists understand computing from a practical, rational, and theoretical perspective, questions about education are inherently human questions. Humans are often impractical, irrational, and difficult to make predictions or proofs about. Computing education researchers are using experimentation and design to demonstrate we can address important questions about how humans come to understand computing, and how we can make it better. Research in computing education will pave the way to make "computational thinking" a 21st century literacy that we can share across the campus.

## Understanding Computing Before Programming

A research theme in the early 1980s was how to design programming languages so they would be more like natural languages. An obvious question, then, is how people specify processes in natural language. Lance A. Miller asked his study participants to specify file manipulation tasks for another person. A task might be "Make a list of employees who have a job title of photographer and who are rated superior, given these paper files." Miller studied the language used in his participants' descriptions.[2]

One of Miller's surprises was how rarely his participants explicitly specified any kind of control flow. There was almost no explicit looping in any of their task descriptions. While some tested conditions ("IF"), none ever specified an "ELSE." He found this so surprising that he gave a second set of participants an example task description, without looping and no ELSE specification. The second set of participants easily executed the task description. When asked what they were doing if the condition was not met, or if data was exhausted, they replied (almost unanimously, Miller reports), "Of course, you just check the next person, or if there are no more, you just go on."

Miller's results predict some of the challenges in learning to program—challenges that are well-known to teachers of introductory classes today. While process descriptions by novices tend not to specify what to do under every condition, computers require that specificity. Miller's results suggest what kinds of programming languages might be easier for novices. Programming languages like APL and MATLAB, and programming tools for children like Squeak's eToys use implicit looping, as did the participants in Miller's studies.

Twenty years later, John Pane and his colleagues at Carnegie Mellon University revisited Miller's questions, in new contexts.[3] In one experiment, Pane showed his subjects situations and processes that occur in a Pacman game, then asked how they would specify them. The subjects responded with explanations like, "When Pacman gets all the dots, he goes to the next level." Like Miller, Pane found that participants rarely used explicit looping and always used one-sided conditionals. Pane went further, to characterize the style of programming that the participants used. He found that over half of the participants' task statements were in the form of production rules, as in the example. He also saw the use of constraints and imperative statements, but little evidence of object-oriented thinking. Participants did talk about accessing behaviors built into an entity, but rarely from the perspective of that entity; instead, it was from the perspective of the player or the programmer. He found no evidence of participants describing categories of entities (defining classes), inheritance, or polymorphism.

Pane's results suggest that object-oriented thinking is not "natural," in the sense of being characteristic of novices' task descriptions. Since ob-

**Figure 1: Traditional conditional structure.**

```
if (value < 10)
then value = value + 10;
else sum = sum + value;
end if
```

**Figure 2: New conditional structure.**

```
if (value < 10): value = value + 10;
not (value < 10): sum = sum + value;
end (value < 10)
```

jects are the foundation of most modern software today, his results point out where we can expect to find challenges in explaining objects to students. Both Miller's and Pane's results encourage us to think how we might design languages for novices that play to their natural ways of thinking about specifying computation, like the use of event-based programming in MIT's Scratch.

In the last four years, a multinational group of researchers has explored "Commonsense Computing": what do our students know before we teach them? Given a complex task, how do people without programming knowledge specify an algorithm for that task? In one paper, Lewandowski et al.[1] explore concurrency—in a complex task of multiple box offices selling tickets for a theater, how well do non-programming students avoid selling the same seat twice? The results showed that 97 solutions (69% of the total, drawn from five institutions) were correct; only 31% of the solutions (45% of the correct solutions) were distributed, so teachers of algorithms classes need not worry about being put out of business. Non-computing students do not naturally come up with the elegant solutions that computer scientists have devised. However, these results suggest that students can "naturally" think about concurrency correctly. Problems with implementing concurrent programs might stem more from the challenges in specifying those algorithms in current programming languages, rather than from the complexity of the algorithms themselves.

## Redesigning Programming Languages

Both Pane's and Miller's results make suggestions about the design of programming languages if the goal is to make computational ideas more accessible to novices. Testing new forms of programming languages was an area of active exploration by Thomas R.G. Green, Elliot Soloway, and others.

In one paper, Green and his colleagues explored alternatives to the traditional conditional structure.[5] A typical structure might look like the structure shown in Figure 1. They tested a new structure where this would be written as shown in Figure 2. This new structure makes explicit the condition for the execution of each clause of the condition. Green and his colleagues found that novices were able to correct mistakes using the second form 10 times faster than programs using the first form.

**Research in computing education will pave the way to make "computational thinking" a 21st century literacy that we can share across the campus.**

Miller and Pane found that their participants simply never used an else clause. Instead, it seemed obvious ("of course") what to do when the tested condition wasn't true. Miller's and Pane's subjects were doing something different than Green's. Writing a task description is different than reading and fixing a task description. Green's results complement Miller's and Pane's. Novices do not naturally write the else clause—they think it's obvious what to do if the test fails. However, conditionals in programs are not always obvious, and it's easier for the novices trying to read those programs if the conditions for each clause's execution are explicit.

## Paving the Way for "Computational Thinking" For All

To make "computational thinking" accessible to students across the entire campus, we need to understand how to teach computing better. Computing education researchers explore how humans come to understand computing, and how to improve that understanding. Computing education research is a close cousin to human-computer interaction, since HCI researchers explore how humans interact with computing and how to improve that interaction. Computing education researchers have found a home in the International Computing Education Research (ICER) workshop (whose fourth annual meeting will be held this September in Sydney, Australia; see www.newcastle.edu.au/conference/icer2008/) and in journals like *Computer Science Education* and *Journal on Educational Resources in Computing*.

Computing education research draws on a variety of disciplines to make computing education better. Social scientists like Jane Margolis, Lecia Barker, and Carsten Schulte help us to understand how students experience our classes (which often differs from what we might expect as teachers) and how we can change our classes to make them more successful for all students. Computing education researchers draw on methods from education, sociology, and psychology in order to measure learning about computing and understand the factors that influence that learning. By making computing education better, we can broaden access to computing ideas and capabilities. When we can teach every student programming and the theory of computation in a way that makes sense to them for their discipline, we will see how ubiquitous understanding of computing will advance the entire academy, just as Perlis predicted over 45 years ago. C

**References**

1. Lewandowski, G. et al. Commonsense computing (episode 3): Concurrency and concert tickets. In *Proceedings of the Third International Workshop on Computing Education Research* (2007), 133–144.
2. Miller, L.A. Natural language programming: Styles, strategies, and contrasts. *IBM Systems Journal 29*, 2 (1981), 184–215.
3. Pane, J.F., Ratanamahatana, C., and Myers, B.A. Studying the language and structure in non-programmers' solutions to programming problems. *International Journal of Human-Computer Studies 54* (2001), 237–264.
4. Perlis, A. The computer in the university. In M. Greenberger, Ed., *Computers and the World of the Future*, MIT Press, Cambridge, MA, 1962, 180–219.
5. Sime, M.E., Arblaster, A.T., and Green, T.R.G. Structuring the programmer's task. *Journal of Occupational Psychology 50* (1977), 205–216.
6. Wing, J. Computational thinking. *Commun. ACM 49*, 3 (Mar. 2006), 33–35.

**Mark Guzdial** (guzdial@cc.gatech.edu) is a professor in the College of Computing at Georgia Institute of Technology in Atlanta, GA.

The *Communications* "Education" column will feature commentary on education issues, presenting research results and opinions that inform how the challenges of computing education can be best addressed.