CHAPTER SEVEN

Why Is It So Hard to Learn to Program?

Mark Guzdial

Most of the topics in this book—the best ways to develop software, the costs of developing software, what communications facilitate programming—revolve in some way around programmers. But it's hard even to become a programmer in the first place. Few people try to enter the field, and even fewer succeed. In this chapter, we ask why it's so hard to learn to program.

Whether we need to develop more programmers at this moment is a point of dispute. The United States Bureau of Labor Statistics recently predicted an enormous demand for computing professionals. According to the November 2008 report [Association 2008], the demand for "IT Professionals" from 2006–2016 will be twice the growth rate of the rest of the workforce. The updated estimate of November 2009 said: "'Computer and mathematical' occupations are the fastest growing occupational cluster within the fastest growing major occupational group" [Consortium 2010]. But what does "IT Professional" mean? A "computer and mathematical" occupation? The experience of many newly unemployed IT workers, especially during the current downturn, suggests that maybe there are *too many* programmers in the United States today [Rampell 2010].

Although it may not be clear whether we *need* more programmers, it is clear that many start down the path of programming and fail early. Rumors of high failure rates in introductory computing courses (typically referred to as "CS1" in reference to an early curriculum standards report) are common in the literature and in hallway discussions at conferences such as the ACM Special Interest Group in Computer Science Education (SIGCSE) Symposium. Jens Bennedsen and Michael Caspersen made the first reasonable attempt to find out what failure

rates really look like [Bennedsen and Caspersen 2007]. They asked for data from faculty around the world via several computer science (CS) educator mailing lists. Sixty-three institutions provided their failure rates in introductory courses, which means that these data are self-selected and self-reported (e.g., schools with really embarrassing results may have chosen not to participate, or to lie, and the results can be skewed because the sampling was not random). Overall, 30% of students fail or withdraw from the first course, with higher rates in colleges than universities (40% versus 30%). Thus, we have indications that roughly one out of every three students who start a CS1 course, around the world in all kinds of institutions, fails or gives up. Why is that?

The Bennedsen and Caspersen results report success or failure in taking a class. A CS1 teacher's criteria aren't the only possible definition of success, however. There are many programmers who never took a course at all, yet are successful. So we first need to establish that students really do have a hard time learning programming, apart from the evidence of grades. If we can establish that, the next question is, "Why?" Is programming an unnatural activity? Could programming be made easier in a different form? Could programming be taught in a different way that makes learning easier? Or maybe we just have no idea how to actually measure what students know about programming.

Do Students Have Difficulty Learning to Program?

At Yale University in the 1980s, Elliot Soloway gave the same assignment regularly in his Pascal programming class [Soloway et al. 1983]:

> Write a program that repeatedly reads in positive integers, until it reads the integer 99999. After seeing 99999, it should print out the average.

Called "The Rainfall Problem," it became one of the most studied problems in the early years of computing education research. In the 1983 paper from which this formulation of the problem is taken (other formulations were explored in other papers), the Yale team was exploring whether having a *leave* statement (a *break* in C or Python) improved student performance on the problem. They gave the problem to three groups of students:

- First-semester CS1 students after learning and using WHILE, REPEAT, and FOR, 3/4 of the way through the term
- CS2 (second semester of computing coursework, typically a data structures course) students 3/4 of the way through the term
- Juniors and seniors in a systems programming course

In each class of students, half the students used traditional Pascal and the other half had the opportunity to use Pascal with an added *leave* statement. The results, summarized in Table 7-1, may shock those who have successfully entered a programming career. Only 14% of the introductory students could solve this problem in raw Pascal? And 30% of the *most advanced* students couldn't solve it either? This study was repeated in the literature several

times (e.g., dissertations by both Jim Spohrer [Spohrer 1992] and Lewis Johnson [Johnson and Soloway 1987] studied students working on the Rainfall Problem), and has been repeated informally many times since—with surprisingly similar results every time.

TABLE 7-1. Performance of Yale students on the Rainfall Problem

Class group	% correct using raw Pascal	% correct using Pascal with *leave*
CS1	14%	24%
CS2	36%	61%
Systems course	69%	96%

The problem requires a relatively complex conditional controlling the loop. *If the input is negative, ignore it but continue to accept input. If the input is positive and not 99999, add it to the total and increment the count. If the input is 99999, ignore the input and leave the loop.* It's easy to get an error where negative input or 99999 gets added into the sum.

These results were at Yale University. Could it be that Yale was just really bad at teaching programming? Few students in those years might have learned programming before entering college, so whatever instruction they received came from the CS1 course. Researchers for many years wondered how to conduct a study of programming that would avoid the complicating factor of possibly bad instruction at a particular school.

The 2001 McCracken Working Group

In 2001, Mike McCracken [McCracken et al. 2001] organized a group of researchers to meet at the Innovation and Technology in Computer Science Education (ITICSE) conference held at the University of Canterbury at Kent. ITICSE is a European conference, and it draws participants from all over the world. Teachers in McCracken's group were to conduct the same study in each of their CS1 or CS2 classes: assign the same problem, and give students 90 minutes to complete the assignment on paper. All the student data were submitted and analyzed by the participants at the conference. Four institutions from three countries participated in this first *multi-institutional, multi-national* (MIMN) study of student performance in computing. By comparing students across institutional and country borders, the researchers hoped to get a clear picture of just what students *really* can do in their first courses.

The problem was to evaluate arithmetic expressions (prefix, infix, or postfix) with only numbers, binary operators (+, –, /, *), or unary negation (~ in order to avoid the complication of overloading the minus character). Overall, 216 students submitted answers. The average score on a language-independent grading rubric of 110 points was 22.89 (21%). Students did *horribly* on this problem. One teacher even "cheated" by explicitly lecturing on expression evaluation before the problem was assigned. That class performed no better.

The McCracken Working Group did several evaluations of their data. They found that performance varied dramatically between classes. They also saw evidence of the "two hump effect" that many computing teachers have noted and several papers have tried to explain. Some students just "get it" and perform very well. The larger hump of students performs much worse. Why is it that some students just "get" programming and others don't? Variables ranging from past computing experience to mathematical background have been explored [Bennedsen and Caspersen 2005], and there is still no convincing explanation for this effect.

The Lister Working Group

Some students may not respond to a particular teacher or teaching style, but why did so many students at different institutions do *so* badly? Are we teaching badly *everywhere*? Are we overestimating what the students should be able to do? Or are we not measuring the right things? Raymond Lister organized a second ITICSE Working Group in 2004 to explore some of these questions [Lister et al. 2004].

The Lister group's idea was that the McCracken group asked too much of students. Giving them a problem required a very high level of thinking to design and implement a solution. The Lister group decided to focus instead on the lower-level abilities to read and trace code. They created a multiple-choice questionnaire (MCQ) that asked students to perform tasks such as reading code and identifying outcomes, or identifying the correct code for the empty gaps in a program fragment. The questions focused on array manipulation. They asked their participants from around the world to try the same MCQ with their students, and to bring their results to ITICSE.

The Lister group results were better, but still disappointing. The 556 students had an average score of 60%. Although those results did suggest that the McCracken group overestimated what students could do, Lister and his group expected much better performance on their questions.

The McCracken and Lister efforts taught researchers that it is hard to underestimate how much students understand about programming in their first course. They are clearly learning far less than we realize. Now, *some* students are learning. But the majority are not. What's so hard about programming that most students can't easily pick it up?

What Do People Understand Naturally About Programming?

Linguists generally agree that humans are "wired" for language. Our brains have evolved to pick up language quickly and efficiently. We are wired specifically for *natural* language. Programming is the manipulation of an artificial language invented for a particular, relatively unnatural purpose—telling a nonhuman agent (a computer) *exactly* what to do. Maybe programming is not a natural activity for us, and only a few humans are able to do the complex mental gymnastics to succeed at this unnatural act.

How might we answer this question? We can try an approach similar to Lister's modification to McCracken's approach: choose a smaller part of the task and focus just on that. To program requires telling a machine what to do in an unnatural language. What if we asked study participants to tell another human being to accomplish some task, in natural language? How would participants define their "programs"? If we remove the artificial language, is programming now "natural" or "commonsense"?

L.A. Miller asked participants in his studies to create directions for someone else to perform [Miller 1981]. The participants were given descriptions of various files (such as employees, jobs, and salary information) and tasks like:

> Make one list of employees who meet either of the following criteria:
>
> (1) They have a job title of technician and they make 6 dollars/hr. or more.
>
> (2) They are unmarried and make less than 6 dollars/hr.
>
> List should be organized by employee name.

Miller learned a lot about what was hard and what was easy for his participants. First, his subjects completed their tasks. He doesn't say that 1/3 of all the subjects gave up or failed, as happens all the time in programming classes. The basic challenge of specifying a process for someone else doesn't seem to be the problem.

A key difference between the natural language solutions to Miller's problems and the programming problems studied by earlier researchers is the structure of the solution. Miller's subjects didn't define iterations, but set operations. For instance, they didn't say, "Take each folder and look at the last name. If it starts with 'G'...." Instead, they talked about doing things "for all the last names starting with 'G'...." Miller was surprised at this: no one ever specified the ending conditions for their loops. Some people talked about testable IF-like conditions, but none ever used an ELSE. These results alone point to the possibility of defining a programming language that is *naturally* easier for novices.

Miller ran a separate experiment where he gave other participants the instructions from the first experiment with the vague loops. Nobody had any problem with the instructions. It was obvious that you stopped when you were done with data. Subjects processed the set; they didn't increment an index.

John Pane took up this exploration again in the late 1990s and early 2000s [Pane et al. 2001]. Pane was explicitly interested in creating a programming language that would be closer to how people "naturally" described processes to one another. He replicated Miller's experiments, but with a different task and different input. He was worried that by providing "files" and asking for "lists," Miller may have been leading the witness, so to speak. Instead, Pane showed the subjects images and movies of videogames, then asked them how they would want to tell the computer to make *that* happen, e.g., "Write a statement that summarizes how I (as the computer) should move Pac-Man in relation to the presence or absence of other things."

Pane, like Miller, found that people weren't explicit about their looping. He went further to characterize the kinds of instructions they wrote in terms of programming paradigms. He found lots of use of constraints ("This one is always doing that"), event-driven programming ("When Pac-Man gets all the dots, he goes to the next level"), and imperative programming. Nobody ever talked about *objects* even once. They talked about characteristics and behaviors of entities in the video game, but no groupings of those entities (e.g., into classes). They never talked about the behaviors from the perspective of the entities themselves; everything was from the perspective of the player or the programmer.

Based on Miller and Pane's experiments, we might claim that people may be able to specify tasks to another agent, but that our current programming languages do not allow people to program the way that they think about the tasks. If the programming languages were made more natural, would the majority of student then be able to program? Could people solve complex problems involving significant algorithms in the more natural language? Would a more natural language be good for both novices' tasks and professionals' tasks? And if not, might students of CS still have to deal with learning the professionals' language at some point?

A group of researchers who call themselves the Commonsense Computing Group have been asking some of these questions. They ask pre-CS1 students to solve significant algorithmic tasks, such as sorting or parallelizing a process, in natural language and before they have learned any programming languages. They find their subjects to be surprisingly successful at these tasks.

In one study [Lewandowski et al. 2007], they asked students to create a process for a theater that decides to have two ticket sellers.

> Suppose we sell concert tickets over the telephone in the following way—when a customer calls in and asks for a number (n) of seats, the seller (1) finds the n best seats that are available, (2) marks those n seats as unavailable, and (3) deals with payment options for the customer (e.g., getting a credit or debit card number, or sending the tickets to the Will Call window for pickup).
>
> Suppose we have more than one seller working at the same time. What problems might we see, and how might we avoid those problems?

Some 66 participants across five institutions attempted to solve this problem—with surprising success! As seen in Table 7-2, almost all the students recognized what the real challenge of the problem was, and 71% came up with a solution that would work. Most of these solutions were inefficient—because they involved a centralized arbiter—so there is still much for these students to learn. However, the point that they could solve a parallel processing problem suggests that the problem of getting students to program may be in the tools. Students may be more capable of computational thinking than we give them credit for.

TABLE 7-2. Number of solutions and problems identified by students (n=66), from [Lewandowski et al. 2007]

Accomplishment	Percent of students
Problems identified:	
• Sell ticket more than once	97%
• Other	41%
Provided "reasonable" solutions to concurrency problems	71%

Making the Tools Better by Shifting to Visual Programming

How do we make the tools better? One obvious possible answer is by moving to a more visual notation. Since David Smith's icon-based programming language *Pygmalion* emerged [Smith 1975], the theory has been that maybe visual reasoning is easier for students. There certainly have been a lot of studies showing that visualizations in general helped students in computing [Naps et al. 2003], but relatively few careful studies.

Then, Thomas Green and Marian Petre did a head-to-head comparison between a dataflow-like programming language and a textual programming language [Green and Petre 1992]. They created programs in two visual languages that had been shown to work well in previous studies and in a textual language that had also tested well. Subjects were shown a visual program or a textual program for a short time, and then asked a question about it (e.g., shown input data or output results). *Understanding the graphical language always took more time.* It didn't matter how much experience the subject had with visual or textual languages, or what kind of visual language. Subjects comprehended visual languages more slowly than textual languages.

Green and Petre published several papers on variations of this study [Green et al. 1991]; [Green and Petre 1996], but the real test came when Tom Moher and his colleagues [Moher et al. 1993] stacked the deck in favor of visual languages. Tom and his graduate students were using a visual notation, Petri Nets, to teach programming to high school students. He got a copy of Green and Petre's materials and created a version where the only visual language used was Petri Nets. Then, Tom reran the study with *himself and his students as subjects*. The surprising result was that textual languages were more easily comprehended again, under every condition.

Are we wrong about our intuition about visual languages? Does visualization actually reduce one's understanding of software? What about those studies that Naps et al. were talking about [Naps et al. 2003]? Were they wrong?

There is a standard method for comparing multiple studies, called a *meta-study*. Barbara Kitchenham describes this procedure in Chapter 3 of this book. Chris Hundhausen, Sarah Douglas, and John Stasko did this type of analysis on studies of algorithm visualizations [Hundhausen et al. 2002]. They found that yes, there are a lot of studies showing significant benefits for algorithm visualizations for students. There are a lot of studies with nonsignificant

results. Some studies had significant results but didn't make it obvious *how* algorithmic visualizations might be helping (Figure 7-1). Hundhausen and colleagues found that *how* the visualizations were used matters a *lot.* For example, using visualizations in lecture demonstration had little impact on student learning. But having students build their own visualizations had significant impact on those students' learning.

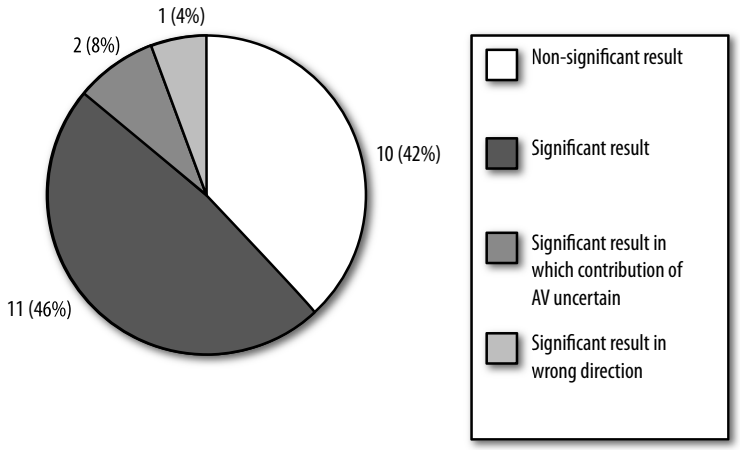

FIGURE 7-1. Summary of 24 studies in Hundhausen, Douglas, and Stasko paper [Hundhausen et al. 2002]

A few studies varied the use of visualization while holding all other variables constant (e.g., type of class, type of student, teacher, subject). Although Hundhausen and colleagues suspect the way visualization is used is important after their analysis of 24 studies, that result is not the same as *testing* the suspicion. One thing we have learned from studies in education is that it is actually quite difficult to predict the outcomes. Humans are not as predictable as a falling projectile or chemical cocktails. We actually have to test our suspicions and hypotheses, and sometimes test them repeatedly under different conditions, until we are convinced about some finding.

Contextualizing for Motivation

What we know thus far is that beginning computing students learn far less about design and programming than we might predict, and they fail their first course at a pretty high rate. We find that changing from textual to visual modalities does not consistently result in better results, which suggests that another variable may be at play. The previous section explored the possibility that differences in how visualizations are used may be one such variable. What other variables might we manipulate in order to improve student success and learning?

In 1999, Georgia Tech decided to require introductory computing of all undergraduate students. For the first few years of this requirement, only one course satisfied this requirement. Overall, the pass rate in this course was 78%, which is quite good by Bennedsen and Caspersen's analysis [Bennedsen and Caspersen 2007]. However, that number wasn't so good when we start disaggregating it. The pass rate for students from the Liberal Arts, Architecture, and Management colleges was less than 50% [Tew et al. 2005a]. Women failed at *nearly twice* the rate of men. A course aimed at *all* students, but at which mostly males in technical majors pass, highlights general problems in teaching computing.

In 2003, we started an experiment to teach students in those classes a different kind of introductory course, one *contextualized* around manipulating media [Forte and Guzdial 2004]. Students worked on essentially the same kinds of programming as in any other introductory computer science course. In fact, we worked hard to cover all the topics [Guzdial 2003] that were recommended by the then-current ACM and IEEE Computing Standards [IEEE-CS/ACM 2001]. However, all the textbook examples, sample code in lectures, and homework assignments had students manipulating digital media as the data in these programs. Students learned how to iterate through the elements of an array, for instance, by converting all the pixels in a picture to grayscale. Instead of concatenating strings, they concatenated sound buffers to do digital splicing. Iteration through a subrange of an array is introduced by removing red-eye from a picture without messing up any other red in the picture.

The response was positive and dramatic. Students found the new course more relevant and compelling, particularly women, whose pass rate rose above men's (but not with any statistical significance) [Rich et al. 2004]. The average pass rate over the next three years rose to 85%, and that applies even to those majors who were passing at less than 50% per semester before [Tew et al. 2005a].

Sounds like a big success, but what do these papers really say? Do we know that the new approach is the *only* thing that changed? Maybe those colleges suddenly started accepting much smarter students. Maybe Georgia Tech hired a new, charismatic instructor who charmed the students into caring about the content. Social science researchers refer to these factors that keep us from claiming what we might *want* to claim from a study *threats to validity*.* We can state, in defense of the value of our change, that the [Tew et al. 2005a] paper included semesters with different instructors, and the results covered three years' worth of results, so it's unlikely that the students were suddenly much different.

Even if we felt confident concluding that Georgia Tech's success was the result of introducing Media Computation, and that all other factors with respect to students and teaching remained the same, we should wonder what we can claim. Georgia Tech is a pretty good school. Smart students go there. They hire and retain good teachers. Could *your* school get better success in introductory computing by introducing Media Computation?

* *http://www.creative-wisdom.com/teaching/WBI/threat.shtml*

The first trial of Media Computation at a different kind of school was by Charles Fowler at Gainesville State College in Georgia. Gainesville State College is a two-year (not undergraduate and post-graduate) public college. The results were reported in the same [Tew et al. 2005a] paper. Fowler also found dramatically improved success rates among his students. Fowler's students ranged from computer science to nursing students. However, both the Georgia Tech and Gainesville students were predominantly white. Would this approach work with minority students?

At the University of Illinois-Chicago (UIC), Pat Troy and Bob Sloan introduced Media Computation into their "CS 0.5" class [Sloan and Troy 2008]. Their class was for students who wanted to major in computer science but had no background in programming. A CS 0.5 class is meant to get them ready for the first ("CS1") course. Over multiple semesters, these students' pass rate also rose. UIC has a much more ethnically diverse student population, where a majority of their students belong to minority ethnic groups.

Are you now convinced that you should use Media Computation with your students? You might argue that these are still unusual cases. The studies at Georgia Tech and Gainesville were with nonmajors (as well as majors at Gainesville). Although Troy and Sloan are dealing with students who want to major in computer science, their class is not the normal introductory computer science course for undergraduate majors.

Beth Simon and her colleagues at the University of California at San Diego (UCSD) started using Media Computation two years ago as the main introductory course ("CS1") for CS majors [Simon et al. 2010]. More students pass with the new course. What's more, the Media Computation students are doing better in the *second* course, the one after Media Computation, than the students who had the traditional CS1.

Is that it? Is Media Computation a slam dunk? Should *everyone* use Media Computation? Although my publisher would like you to believe that [Guzdial and Ericson 2009a]; [Guzdial and Ericson 2009b], the research does not unambiguously bear it out.

First, I haven't claimed that students learn the same amount with Media Computation. If anyone tells you that students learn the same amount with their approach compared to another, be very skeptical, because we currently do not have reliable and valid measures of introductory computing learning to make that claim. Allison Tew (also in [Tew et al. 2005a]) first tried to answer the question of whether students learn the same in different CS1s in 2005 [Tew et al. 2005b]. She developed two multiple-choice question tests in each of the CS1 languages she wanted to compare. She meant them to be *isomorphic*: a problem meant to evaluate a particular concept (say, iteration over an array) would be essentially the same across both tests and all languages. She used these tests before and after a second CS course (CS2) in order to measure how much difference there was in student learning between the different CS1s. Tew found that students *did* learn different things in their CS1 class (as measured by the start-of-CS2 test), but that those differences disappeared by the end of CS2. That's a great

finding, suggesting that the differences in CS1 weren't all that critical to future success. But in future trials, she never found the same result.

How could that be? One real possibility is that her tests *weren't* exactly the same. Students might interpret them differently. They might not measure exactly the kind of learning that she aimed to test. For instance, maybe the answer to some of the multiple-choice questions could be guessed because the distractors (wrong answers) were so far from being feasible that students could dismiss them without really knowing the right answer. A good test for measuring learning should be *reliable* and *valid*—in other words, it measures the right thing, and it's interpreted the same way by all students all the time.

As of this writing we have no measure of CS1 learning that is language-independent, reliable, and valid. Tew is testing one now. But until one exists, it is not possible to determine for sure that students are learning the same things in different approaches. It's great that students succeed more in Media Computation, and it's great that UCSD students do better even in the second course, but we really can't say for sure that students learn the same things.

Second, even if Georgia Tech, Gainesville, UIC, and UCSD were all able to show that students learned the same amount in the introductory course, what would all that prove? That the course will work for *everyone*? That it will be better than the "traditional" course, no matter how marvelous or successful the "traditional" course is? For every kind of student, no matter how ill-prepared or uninterested? No matter how bad the teacher is? That's ridiculous, of course. We can always imagine something that could go wrong.

In general, curricular approaches offer us *prescriptive* models, but not *predictive* theories. Media Computation studies show us evidence that, for a variety of students and teachers, the success rate for the introductory course *can* be improved, but not that it inevitably *will* improve. Being able to promise improvement would be a prescription. It is not a predictive theory in the sense that it can predict improvement, not without knowing many more variables that have not yet been tested with studies. It also can't be predictive because we can't say that *not* using Media Computation *guarantees* failure.

Conclusion: A Fledgling Field

Computing Education Research, as a field of study, is still a fledgling discipline [Fincher and Petre 2004]. We are just recently realizing the importance of computing education and the need to support the teaching of computing. The ACM organization that supports computing teachers, Computer Science Teachers Association (CSTA), was formed only in 2005. In contrast, the National Council of Teachers of Mathematics was formed in 1920.

Most of our studies point more toward how complex it is for humans to learn how to program a computer. We continue to marvel at the intelligence and creativity of humans, and that even students without training in computing can already think in algorithmic terms. However, developing that skill to the point that it can be used to give directions to a machine in the

machine's language occurs more slowly than we might expect. We can get students through the process, but we still don't have effective measures of how much they're learning.

What we really need as a field are predictive theories, based on models of how people really come to develop their understanding of computing. On those theories, we can build curricula in which we can have confidence. ACM's International Computing Education Research workshop is only five years old this year, and the number of attendees has never topped 100. We have few workers, and they have only just started at a difficult task. We have taken only the first steps toward understanding why it is that students find it so hard to learn to program.

References

[Bennedsen and Caspersen 2005] Bennedsen, J., and M.E. Caspersen. 2005. An investigation of potential success factors for an introductory model-driven programming course. *Proceedings of the first international workshop on computing education research*: 155–163.

[Bennedsen and Caspersen 2007] Bennedsen, J., and M.E. Caspersen. 2007. Failure rates in introductory programming. *SIGCSE Bull.* 39(2): 32–36.

[Computing Community 2010] Computing Community Consortium. 2010. Where the jobs are.... *http://www.cccblog.org/2010/01/04/where-the-jobs-are/*.

[Computing Research 2008] Computing Research Association. 2008. BLS predicts strong job growth and high salaries for IT workforce. Retrieved May 2010 from *http://www.cra.org/resources/crn-archive-view-detail/bls_predicts_strong_job_growth_and_high_salaries_for_it_workforce_through_2/*.

[Fincher and Petre 2004] Fincher, S., and M. Petre. 2004. *Computer Science Education Research*. New York: RoutledgeFalmer.

[Forte and Guzdial 2004] Forte, A., and M. Guzdial. 2004. Computers for Communication, Not Calculation: Media As a Motivation and Context for Learning. *Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04)* 4: 40096.1.

[Green and Petre 1992] Green, T.R.G., and M. Petre. 1992. When visual programs are harder to read than textual programs. In *Human-Computer Interaction: Tasks and Organisation, Proceedings EECE-6 (6th European Conference on Cognitive Ergonomics)*, ed. G.C.v.d. Veer, M.J. Tauber, S. Bagnarola, and M. Antavolits, 167–180. Rome: CUD.

[Green and Petre 1996] Green, T.R.G., and M. Petre. 1996. Usability analysis of visual programming environments: A "cognitive dimensions" framework. *Journal of Visual Languages and Computing* 7(2): 131–174.

[Green et al. 1991] Green, T.R.G., M. Petre, et al. 1991. Comprehensibility of visual and textual programs: A test of "superlativism" against the "match-mismatch" conjecture. In *Empirical Studies of Programmers: Fourth Workshop*, ed. J. Koenemann-Belliveau, T. Moher, and S. Robertson, 121–146. Norwood, NJ, Ablex.

[Guzdial 2003] Guzdial, M. 2003. A media computation course for non-majors. *ACM SIGCSE Bulletin* 35(3): 104–108.

[Guzdial and Ericson 2009a] Guzdial, M., and B. Ericson. 2009. *Introduction to Computing and Programming in Python: A Multimedia Approach*, Second Edition. Upper Saddle River, NJ: Pearson Prentice Hall.

[Guzdial and Ericson 2009b] Guzdial, M., and B. Ericson. 2009. *Problem Solving with Data Structures Using Java: A Multimedia Approach*. Upper Saddle River, NJ: Pearson Prentice Hall

[Hundhausen et al. 2002] Hundhausen, C.D., S.H. Douglas, et al. 2002. A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages and Computing* 13: 259–290.

[IEEE-CS/ACM 2001] IEEE Computer Society and Association for Computing Machinery, The Joint Task Force on Computing Curricula. 2001. Computing curricula 2001. *Journal of Educational Resources in Computing.* 1(3): 1.

[Johnson and Soloway 1987] Johnson, W.L., and E. Soloway. 1987. PROUST: An automatic debugger for Pascal programs. In *Artificial intelligence and instruction: Applications and methods*, ed. G. Kearsley, 49–67. Boston: Addison-Wesley Longman Publishing Co., Inc.

[Lewandowski et al. 2007] Lewandowski, G., D.J. Bouvier, et al. 2007. Commonsense computing (episode 3): Concurrency and concert tickets. *Proceedings of the third international workshop on computing education research*: 133–144.

[Lister et al. 2004] Lister, R., E.S. Adams, et al. 2004. A multi-national study of reading and tracing skills in novice programmers. *Working group reports from ITiCSE on Innovation and technology in computer science education*: 119–150.

[McCracken et al. 2001] McCracken, M., V. Almstrum, et al. 2001. A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *SIGCSE Bulletin.* 33(4): 125–180.

[Miller 1981] Miller, L.A. 1981. Natural language programming: Styles, strategies, and contrasts. *IBM Systems Journal* 29(2): 184–215.

[Moher et al. 1993] Moher, T.G., D.C. Mak, et al. 1993. Comparing the comprehensibility of textual and graphical programs: the case of Petri nets. In *Empirical Studies of Programmers: Fifth Workshop*, ed. C.R. Cook, J.C. Scholtz, and J.C. Spohrer, 137–161. Norwood, NJ, Ablex.

[Naps et al. 2003] Naps, T., S. Cooper, et al. 2003. Evaluating the educational impact of visualization. *Working group reports from ITiCSE on innovation and technology in computer science education*: 124–136.

[Pane et al. 2001] Pane, J.F., B.A. Myers, et al. 2001. Studying the language and structure in non-programmers' solutions to programming problems. *International Journal of Human-Computer Studies.* 54(2): 237–264.

[Rampell 2010] Rampell, Catherine. "Once a Dynamo, the Tech Sector Is Slow to Hire." *The New York Times*, Sept. 7, 2010.

[Rich et al. 2004] Rich, L., H. Perry, et al. 2004. A CS1 course designed to address interests of women. *Proceedings of the 35th SIGCSE technical symposium on computer science education*: 190–194.

[Simon et al. 2010] Simon, B., P. Kinnunen, et al. 2010. Experience Report: CS1 for Majors with Media Computation. Paper presented at ACM Innovation and Technology in Computer Science Education Conference, June 26–30, in Ankara, Turkey.

[Sloan and Troy 2008] Sloan, R.H., and P. Troy. 2008. CS 0.5: A better approach to introductory computer science for majors. *Proceedings of the 39th SIGCSE technical symposium on computer science education*: 271–275.

[Smith 1975] Smith, D.C. 1975. PYGMALION: A creative programming environment. Computer Science PhD diss., Stanford University.

[Soloway et al. 1983] Soloway, E., J. Bonar, et al. 1983. Cognitive strategies and looping constructs: An empirical study. *Communications of the ACM* 26(11): 853–860.

[Spohrer 1992] Spohrer, J.C. 1992. *Marcel: Simulating the novice programmer*. Norwood, NJ: Ablex.

[Tew et al. 2005a] Tew, A.E., C. Fowler, et al. 2005. Tracking an innovation in introductory CS education from a research university to a two-year college. *Proceedings of the 36th SIGCSE technical symposium on computer science education*: 416–420.

[Tew et al. 2005b] Tew, A.E., W.M. McCracken, et al. 2005. Impact of alternative introductory courses on programming concept understanding. *Proceedings of the first international workshop on computing education research*: 25–35.