CS Education

Guzdial's Blog, October 5 – November 7, 2007

Why is assessing learning about computing so hard?
7:36 AM PDT, October 5, 2007

I am going to teach a graduate course on Computing Education Research next semester, which I'm very excited about. In discussing the course with a colleague, I was asked, "Where is the paper that lists the 'Hot Questions in CS Ed Research'?" I had to admit that I don't know of one. The book by Sally and Marian, Computer Science Education Research , discusses research questions in the back chapters, but is more a book about methods.

So, I've been giving a lot of thought to our hot research questions, and I expect that I'll describe some of them here.

One of them has to be "How do we assess knowledge about computing?" There are several efforts to come up with new assessments or to create a structure for assessments. The first thing to consider is why assessing knowledge about computing is different and harder than in other fields.

Let's imagine that you've got two or more ways to teach Physics, and you want to compare them. It's a pretty straightforward thing--you teach students in each approach, and then ask them some question about a situation they're familiar about. You ask students to explain how long it will take an object to fall to the ground from the top of a three story building, or you ask them to explain why a cannonball shot into the air eventually falls to the earth--and maybe where that cannonball will land. Therein lies the crux of the problem for computer science. We all know about the physical world. Physics is the science of the physical world. Different approaches just give us different lenses on the same experiences that we have in the physical world. Chemistry, biology, and all the sciences, and all of engineering, has that leg up on us in computer science--the students have real experiences in those domains that they can draw upon.

That's not true for computer science. We don't have a physical experience of computing. It's an interesting and open research question whether use of application software or video games gives people a sense of what a computer does, how it does it, and what the limitations are of what it can do. In a computer science class, especially an introductory course, the students (in most cases) just know the language you taught them. If you only know Java (or Scheme or Alice), it's a rare student who can abstract away the language to talk about what computing can do in general. How could students do that, if their only experience of computing is through the one language you taught them? Computer science teachers complain about tests that are specific to a language or seem hung up on the syntax of a given language--but how can you do otherwise if that language is the only way that the students understand computing at all?

Maybe that's why learning a second language is frequently harder than learning the first language (a phenomenon first described by, to the best of my knowledge, Ben Shneiderman in the 1980's). Maybe you mostly just learn the language the first time. With the second language, you have two lenses on the topic of computing, and now you really are learning computer science. You really can abstract away the language and start thinking about what the computer as a powerful and still abstract (though slightly more concrete) entity can do, how it works, and what its limitations are. Maybe the reason that the second language is so hard is that that's really your first experience with computer science, and you are struggling with what a computer is.

In general, we don't know how to assess what students know about computing, apart from any programming language, if they only know a single programming language. That programming language is their only definition of what a computer is. Can we teach computing without a programming language? Mathematicians did. Turing defined what a computer is, without a programming language--by defining a language. Can someone learn what a computer is, how it works, and what its limitations are simply through non-programming activities? There's another open research question.

Course announcement: Computing Education Research
7:52 AM PDT, October 8, 2007

Thought you readers might be interested in how we're describing the field of computing education research in the course announcement I'm sending around here at Georgia Tech:

CS8803: Computing Education Research

To be offered Spring 2008

Computing education research is the study of how people come to understand computational processes and devices, and how to improve that understanding. As computation becomes ubiquitous in our world, understanding of computing in order to design, structure, maintain, and utilize these technologies becomes increasingly important--for the computing professional, for the professional who relies on computing, and for the computationally literate citizen. The research study of how the understanding of computing develops, and how to improve that understanding, is critically important for the technology-dependent societies in which we live.

The course may be useful to computing students interested in academic teaching, though the course is not about techniques and practices of computing education. This course will focus on five questions particularly:

1. How does computing education research compare to other domain-specific education research, such as math education and science education?
2. What are the challenges from an education perspective for computing education? These range from the difficulty of evaluating computing learning and of the broad lack of teacher certification in a time of "No Child Left Behind," to why pointers and recursion are such difficult concepts.
3. What are the affective dimensions of computing education, e.g., how are people motivated and engaged in understanding technology -- or not?
4. What is the relation between learning about computing and learning other disciplines? Can knowledge of computing facilitate learning other knowledge?
5. What role might technology play to facilitate learning about computing?

What makes programming so hard?
5:53 AM PDT, October 18, 2007, updated at 8:08 AM PDT, October 18, 2007

One of most persistent questions in computing education is the reason for the 20% Rule. In every introduction to programming course, 20% of the students just get it effortlessly -- you could lock them in a dimly lit closet with a reference manual, and they'd still figure out how to program. 20% of the class never seems to get it. Why is that?

In this blog previously, I've dismissed the notion of a "geek gene." It can't be that it's nature -- there must be some experience or reflections that those who "get it" have had, and those that don't have not had. A parallel question to why there's a difference is why is it so hard in the first place. What makes

programming different from our everyday life experiences? (Presuming that 100% of every class has made it at least into their teens, they're pretty successful at everyday life.)

My advisor, Elliot Soloway, addressed that question with several of his students. Jeffrey Bonar claimed that it was the interaction between natural language and programming languages -- there were assumptions from natural language that are false in programming languages. Jim Spohrer came up with the pithy phrase that still gets mentioned at computing education research conferences -- "It's composition, not decomposition." People can figure out the pieces of the solution that they need from the programming language (decomposition). It's assembling those pieces into a working program (composition) that's so hard.

Recently, a group of researchers (including Beth Simon and Gary Lewandowski) have been conducting "commonsense computing" experiments. Take someone who knows nothing about programming, and have them explain how to do some algorithmic task, like sorting. The question they're asking is, "Can people come up with algorithms without knowing about programming? Is the challenge of programming the definition of the algorithm?" The paper that they presented at ACM ICER 2007 was their best yet. They asked people to solve a concurrency problem -- two ticket booths want to sell out a theater without ever selling the same seat twice. The vast majority of people came up with perfectly workable solutions. The solutions weren't necessarily the most efficient -- that proves that computer science really has something to teach people about algorithms. The paper did convince me that people can invent workable algorithms.

So, it's not the algorithm. Maybe it's the language -- maybe Bonar was right. John Pane asked that question in his dissertation work at CMU with Brad Myer. He showed people parts of a videogame, then asked them to tell him what they thought that someone would tell a computer to do to make that part of the videogame. Here comes the cool part: John then built a programming system where people could specify the videogame the way that his subjects described it to him! John could then be sure that his programming system matched the language that people wanted to use. The end result was disappointing. While John learned alot about what language features people wanted, his subjects still couldn't program much better than a traditional CS1. I think Pane answered Bonar -- the language may play a role, but it's not the main thing.

Where does that leave us? Here are two hypotheses still outstanding about what makes programming so hard.

The first is that it's the puzzle nature of programming. We've all seen puzzles like "Here are 9 dots -- cross all of them with four lines" or "Here's a map with one-ways on it -- get from point A to point B using only two left turns and three right turns." We all know how to make lines and make turns. It's the puzzle of matching exactly those pieces in just the right way to make it all work. This is essentially Spohrer's claim -- we know the pieces, it's about putting them together that's hard.

At our CS Ed Research seminar last week, Brian Dorn suggested another one. It's the specificity of a program, the need for exactness when our natural world allows for ambiguity. Natural language did not evolve to specify video games or algorithms. Natural language evolved to allow interaction between thinking beings. Programming languages are about specifying a process to a machine. Dorn's hypothesis would say that the problem of Commonsense Computing Researchers and John Pane is that what we describe in natural language is necessarily ambiguous, and the process of getting it exactly right for the machine is the hard part. The example that Mike Hewner in our group provided was that he bets that the Commonsense Computing subjects did not get right the number of iterations necessary in a sort to make sure that the list is truly sorted -- no more, no less.

This is one of the biggest challenges in computing education research, with big impacts. If we knew why programming was so hard, maybe we'd also have some insight into why some programmers are

two or more magnitudes better than others (claim made in Fred Brooks' Mythical Man Month and supported by others since). Solving both of these problems would be a huge advance for our research community and would have direct impact on education and practice in computing.

What value is visualization in programming or software design?
11:01 AM PDT, October 20, 2007

People have wondered for at least 30 years if visualization could make programming easier, at least for novices. Alan Kay suggested this possibility in some of his early writings about Smalltalk and user interfaces. His student, David Canfield Smith, created the first visual programming language in Smalltalk-72, Pygmalion. Smith actually invented the first icons on a computer screen, and later went on to head up the team who developed the Xerox Star, the first commercial personal computer with our familiar desktop user interface.

For decades after, various teams tried to create a visual programming language that was easier to use, learn, and/or debug than a textual programming language. Marian Petre and Thomas Green did the experiments that led to laying that issue to rest. They ran an experiment where subjects were shown a program in one of three different kinds of visual languages or in a textual language. They asked subjects to answer questions about the programs or to find a bug. In every case, textual programming beat out all the visual languages.

For me, the real clincher was Tom Moher's experiment that was later published in the Empirical Studies of Programmers (ESP) Workshop proceedings (still the source for the best research on studying programming). Moher was teaching programming to high school students using the visual notation of Petri Nets. He got Petre's materials and converted them to Petri Nets. Then he stacked the deck—the only subjects he ran were himself and his graduate students who were developing and teaching Petri Nets. Text still won every test. I was at that ESP and remember the amazed response.

Marian Petre wrote a couple of great papers summarizing that work. The bottomline was that visual programming languages required learning to use them, just like textual languages. Visual does have some affordances—no one would do VLSI without visual CAD tools, for example, Text has an advantage in that most people learn complex text structures in written natural language, but might not learn to use handle the same complex structures in visual notations by the time that they learn to program.

At ICER 2007, John Stasko gave a great keynote talk summarizing what we know about using visualizations to teach algorithms. In general, over 15 years of research have shown few benefits to the use of animations for explaining the workings of algorithms for sorting, searching, and even more complex algorithms like priority queues and tree balancing. (There is a chapter by John Stasko and Chris Hundhausen summarizing algorithm animations work in the book Computer Science Education Research by Sally Fincher and Marian Petre.) Mostly, algorithm animations seem to help the most advanced student. If you are a struggling student who isn't quite understanding the textual notation for an algorithm, adding another visual symbolic notation on top of the more-familiar text does not make it better or easier. The text-struggling student now has to struggle with what the visuals mean, too. On the other hand, the better students who understand the text get additional insight from the visuals.

John did point out a really important issue that came up in his studies, which I think points to something generally important. He found something cool when he stopped trying to conduct controlled experiments ("Group A will look at the animation just as long as Group B is studying the text") and just let students use the animations as they wish—the students used the animation for a long time. They liked the animations. They did end up learning from the animations, partially because they spent so much time playing with them.

There is intuitively something related in programming knowledge and visualizations. Consider flowcharts and UML and the whole history of design notation—why do we keep inventing visual depictions of our software and our designs? Why did John's students want to spend so much time with the animation? What were they getting out of them? Why was it that the better students did get some new insights out of algorithm visualizations? Maybe our mistake is thinking that the visualizations have any use for novices. Maybe suitably advanced and abstracted computing knowledge is somehow related to how our brains process images and diagrams, and that's why we keep finding our field reinventing visual notations for software. It may be that we just haven't figured out yet what that connection is and learned how to exploit it to improve software design and development—for the experienced computer scientist, not the student.

What is the role of language in learning programming? (Part 2 of "What makes programming so hard?")
7:14 AM PDT, October 25, 2007

Many people responded to my entry "What makes programming so hard?" I appreciate all the interest and the feedback! My colleague, Blair MacIntyre, gave me some particularly lovely and pointed feedback. He told me that he liked the blog entry, and that I made research in computing education interesting, even compelling. He felt like he might want to do work in this area. However, he also told me that I make the research too scary, the questions too hard to answer. So, let me lay out some more accessible questions--the next questions to answer, the ones that lead towards answering the big questions.

John Pane wrote me and provided me with some additional information about his dissertation work that I had missed in my summary. His subjects in his study were only 10 years old, and he only gave them 1/2 day of instruction in his video game system. So, his results aren't such an obvious answer to the question of the role of language in learning programming. His dissertation work then suggests a bunch of other questions that we might want to ask (some of which John has explored in his later research):

- Maybe 10 year olds are too young, not having proceeded far enough up the Piagetian steps toward abstract thinking. Could adults do better?
- Is 1/2 day of instruction enough? We might expect someone to learn a word processing system with 1/2 day of instruction. However, programming has more inherent complexity. Perhaps we might compare it to something like constructing a spreadsheet. Can someone learn more about programming using a well-designed system like John's in 1/2 day than a comparable someone might learn about spreadsheets in the same 1/2 day?
- John had to make a bunch of design decisions in the construction of his user interface. Maybe there are better design decisions. Exploring variations of his user interface might have different results.

All of these questions lead to the general question of the role of language in learning programming. Certainly, there is some role. Eric Roberts of Stanford has observed that the downturn in interest and enrollment in CS came at the same time that the predominant introductory language in the U.S. moved from Pascal to C++ and then Java. (One phrase that I heard at the ACM SIGCSE conference a couple years ago: "How bad was using C++ in intro courses? It was so bad that Java looked better!") Most computing teachers who have worked over these 20 years will tell you that C++ and Java are clearly harder for students to use than Pascal. Measuring that additional complexity is a significant challenge (see earlier blog posting on the challenge of assessing learning about programming). Part of the problem is syntactic. C++ and Java require more typing and have more syntactic rules than earlier intro languages (Pascal, and also Logo or even Basic). There's another level of complexity, though -- the model of computation that a language requires the programmer to use.

I have argued previously in this blog that objects are hard, that object-oriented programming is harder than procedure or function oriented programming. I will address the role of paradigm in introductory programming in a later blog entry. Even leaving that alone, it's pretty clear that "public static void main(String [] args)" requires a lot of explanation, and that's where you start in Java. That's a lot of inherent cognitive complexity, even if the syntax were easier.

Experience with Alice is an example of how cognitive complexity trumps syntactic complexity. Alice, as I'm sure most of you readers know, is a visual programming environment for storytelling. We teach Alice and Scratch (another visual language), as well as Python and Java to computer science teachers in our workshops. Alice has no syntax to type--you simply drag and drop the language components. You can't get a syntax error.

However, we found that it takes teachers longer to get started with Alice than with Python. We can get teachers to write their own Python Media Computation code in half a day--we'll start in the morning, and teachers will code some fun image manipulations at our lunch break. Alice has a longer learning curve. No, there are no syntax errors, but if you try to drag the wrong thing into the wrong place, it won't drop--and you won't get any messages explaining why. You can't use Alice unless you understand the underlying computational model. You simply can't assemble pieces appropriately without knowing that model and assembling according to that model. It's not a simple model, and that's where most of the effort goes in learning Alice.

And yet, teachers love Alice, and seem more likely to adopt it for their classes than our Python Media Computation. Why? That's part of Lijun Ni's research, which I've mentioned previously in this blog -- and there's more to talk about there, as well. Let me suggest here one big part of it. Alice is about storytelling, and we teach Python for media computation. Storytelling is an even bigger motivator, an even more fundamental driver of human behavior, than manipulating media. I continue to believe that the most significant bit in helping people learn programming is the motivation. What are you doing with programming? The language can get in the way, but people will go through a huge amount of effort to do something that they want to do.

What is the best paradigm for introducing programming?
2:16 PM PST, November 6, 2007

This question leads into one of the most heated arguments in computer science education today. There are the TeachScheme folks who argue strongly for a functions-first approach, teaching Scheme without mutable variables (at least, at first). Then there is the intense “Objects First!” versus “Objects Late!” debate, best characterized by the debate “Resolved: Objects First has Failed” at SIGCSE 2005.

The first problem in answering the question is determining what “best” (and “failed”) means. One interpretation is “What paradigm most closely matches the way that novices think?” My interpretation of the available evidence is that a declarative approach matches most naturally how novices want to specify process and data structures to a computer. For example, the work on SQL in the 1970’s supported the claim that users more easily understood statements like “for all records where the last name starts with ‘G’…” as opposed to “for all records, if the last name starts with ‘G’…"

However, if your goal is to teach students about algorithms, how to write algorithms, and how to solve problems with algorithms, it’s pretty clear that a procedural approach has the smallest cognitive load, meaning that students can focus on algorithms without extra, unecessary stuff like public static void main. While I don’t know evidence supporting this, my experience is that declarative approaches have a limit--at some increased level of complexity, it becomes harder to specify what the computer it is to do than it is in a procedural language. I don't know how one measures or verifies this claim.

Supporters have argued that a functional approach is most like mathematics (and is thus more familiar)

and that non-mutable variables lead to more easily maintained software, are easier to understand, and are more likely to be provable. A paper at ESP by Michael Eisenberg and Mitchel Resnick showed how hard functional programming is for students. If the students are comparing functional programming to mathematics, it's not obvious and even if it's happening, it's not clear that it's helping. That's an interesting avenue for research, though.

The issue of variable mutability (whether you can assign values to variables, versus non-mutable variables where making all computation work through functions and parameters) is a red herring. Yes, there is plenty of evidence that understanding traditional, mutable variables is a challenge for students. There is no evidence that avoiding mutable variables makes programming better. There is evidence that understanding mutable variables is key to learning programming. Saeed Dehnadi and Richard Bornat published a paper "The Camel has Two Humps" that claimed to have a test for success in CS1 that was completely accurate, and was mostly about understanding how mutable variables work. Later attempts to replicate their finding have failed, and now they themselves have pointed out that their test is no longer perfectly predictive. However, even the failed attempts point to the importance of understanding mutable variables for later success in CS. Those are just correlations, not causations, though. Is it necessary to understand mutable variables to continue on in computer science? What if the entire curriculum were non-mutable? Those are open questions, but the available evidence doesn't make them promising.

The biggest question is the role of object-oriented programming. Objects do encapsulate complex computational behavior and data, and the available evidence (from Alice, Squeak eToys, Media Computation, and other sources) suggest that working with objects (instantiating them, calling methods on them) does make the creation of complicated programs/worlds/artifacts easier for novice students. However, creating classes or new kinds of objects and figuring out how to change existing classes has always been hard for students. Adele Goldberg's memos about Smalltalk in the mid-1970's document that even their students couldn't figure out which object had the method that they wanted. John Carroll and Mary Beth Rosson's work a decade later identified the problems that professional programmers had in distinguishing between classes and instances. There is no evidence that full object-oriented programming (including class modification and creation) makes anything easier for students new to programming.

Students aiming to be professional software developers must learn object-oriented development eventually. It is the current practice. There is lots of evidence that students can learn procedural programming first and object-oriented development later—virtually every object-oriented developer past the age of 35 did exactly that. Humans are amazingly plastic. They learn easily and often. The notion that students must learn objects first or forever be tainted has no support, but would make for a fascinating research study.

In the end, it's pretty clear that the right answer to this question is "None of the above" or maybe "All of the above." Procedural programming with object use seems to be the best current practice for making it easy to get started programming, and does not seem to hurt later student learning. That ignores student innate ability and interest in declarative approaches though. Mixed models are the most promising future direction. By whatever definition you might have for "best," we're not there yet.

Why are pointers and recursion so hard?
3:40 PM PST, November 6, 2007

When I was a graduate student in Education classes, I was told that we should think about Education as "Psychology Engineering." Education uses what the science of Psychology tells us about how people learn and how to make that work better. The relationship between Education and Psychology, though, is like any related science and engineering discipline. Science provides the theory and explanations for

what engineering does, but it's sometimes the case (used to be often the case) that engineering does things before the science catches up. For example, Ancient Egyptians built the pyramids before science of friction and levers was well understood.

Why is it hard for students to learn pointers and recursion? This is a case where the engineers (computer science teachers) are ahead of the scientists (learning scientists, or computer science education researchers). Every computer science teacher knows that pointers and recursion are hard. There are gobs of ACM SIGCSE papers that talk about these problems and invent solutions.

Some people call these kinds of problematic topics “Threshold Concepts” -- if you “get it,” you pass a threshold where you think about the field or the topic differently. Robert McCartney and his colleagues had a nice paper at ICER 2007 on these topics called “From Limen to Lumen: Computing students in liminal spaces.” "Liminal" here refers to being like a "door," that when you pass through, it's a different world.

But why are they a problem? There's lots of cognitive science and learning science theory that I don't know. What I do know doesn't explain why they are so hard for students to learn. In general, what we know about learning (at the neuronal level, at the conceptual level, in terms of associations and productions rules and cases) doesn't distinguish between what is being learned—learning is learning.

That's the general form. In specific, learning that forces accommodation (in Piagetian terms) or conceptual change (as Chi or diSessa would call it) is harder. Learning something that forces you to re-think what you knew before is more challenging than simple accretion of new knowledge (what Piaget called assimilation).

Why are recursion and pointers “liminal” or “threshold” concepts? Maybe recursion forces students to rethink what they thought about iteration or maybe algorithms more generally. Maybe pointers force students to re-think how variables work. This seems to be a ripe area for future research.

Can students learn introductory computing from pseudocode?
3:54 PM PST, November 6, 2007

E.W. Dijkstra wrote a famous paper, published in "Communications of the ACM" on "The Cruelty of Really Teaching Computer Science." He argued that programming in the first semester was a bad idea, that you only really learned to program by not using a computer. He believed that one had to think about a program's execution before allowing oneself the luxury of using a computer to demonstrate or prove how the program would execute. That was real computer science.

Of course, humans can learn to program like that. Mathematicians invented programming without computers to test it upon. Turing didn't really have a machine on a read-write tape when he figured out what that machine might do. The issue is, do most students learn like that?

In some sense, education is like public health. Of course, some people will do the right thing, will get their flu shots, will watch their weight, and will wash their hands often. There would be far fewer challenges to public health if all people did the right thing. They don't. So, how do we improve the health of the whole public? By cajoling and tricking and convincing people to do the right things. That's the case for education, too.

Yes, some people can learn computing without a computer, as Dijkstra and Turing did. Most people will not. Most people need to have the luxury of a way to test their hypotheses. The computer is what should make introductory computing easier to learn than mathematics--we have a way for students to test their work. Do we know how to use the technology well so that works? Is it really easier to learn computer science than mathematics? (How would you know?) Can we teach students to learn computer science as well as the pseudocode approach, while using computers? Maybe that last question does not

even make sense. Can we help people to have better health, even if they don't do the "right things"? Probably not.

Dijkstra's probably right. Learning computing without computers probably would lead to the best understanding of computing--but very few people will be able to do it. Can we afford that?

What is computer science education research?
9:00 AM PST, November 7, 2007

I started this series of blog entries because my colleague, Beki Grinter, asked me what are the hot research questions in computer science education research. This question became even more relevant for me last week when the chair of the School of Interactive Computing gave a talk on "What is Interactive Computing?" This talk raised the question for me, my students, and my colleagues: What is computer science education research, and does it fit into computer science departments, or our new Ph.D. in Human-Centered Computing, or a School of Interactive Computing?

One way of defining a research field is through the questions that one might ask in that field. That's what I've tried to do in these last few blog entries. One can also define a research area more explicitly. Computer science education research is the study of how people come to understand computing and how to improve that understanding. A definition like that needs to be unpacked some to make sense.

Part of computer science education research is about how to change computing education to make it work better, or to achieve particular goals (e.g., broader participation). That kind of improvement might involve changing the technology, or the pedagogy (how concepts are taught or explained), or the curriculum. Clearly, those kinds of changes belong in computer science departments--that's figuring out what we do for our own. Just making the change is not research, though. Computer science education research is where theory or empirical evidence is used to make design decisions (of technology, pedagogy, or curriculum), and then evaluation is used to test those decisions, so that the results inform the community of researchers.

The part that I find more interesting and more challenging is the first part of my definition, the study of how people come to understand computing. Let me explain what that is through an analogy. Computer science education research is related to computer science theory in the same way that psychophysics is related to physics. Physics is the study of the matter in the universe and how it works. Psychophysics is the study of how humans perceive that physical reality. We don't perceive reality exactly as it is. We hear piano keys as being "linear" in some sense, though physicists will tell you that the frequency of each octave rises exponentially. Because of the way that our ears work, we see two orange spots in the rainbow because we can't distinguish between spectral orange and a blended orange.

Computer science education research is concerned with how people make sense of computing reality. Computer science theorists define what computing is. What theory defines is probably not what people perceive.

We in CS Ed research have a different kind of problem than the psychophysicists. The reality of physics is pre-defined and is not generally malleable. Physics is about understanding that reality. Computing reality is malleable. There are rules of computation--we don't know them all (most likely), but there are things that are as fixed as physical laws. For the most part, however, we can construct a reality that is focused on the user's task and allows them to ignore the rules and details of non-task-related computing. HCI is all about creating sensible computing realities within those rules. Those realities overlay and replace for the user the reality that computer scientists see. Microsoft Word, PowerPoint, and manipulating folders of files are a far cry from binary, cache memories, and inodes.

At some point, though, the hard reality of computation shines through, even in those well-designed, user-centered, task-centered worlds. That's where CS Ed Research takes over. What stories do users tell

themselves to explain why graphical objects don't render correctly, or how viruses make pop-up windows appear in Internet Explorer, or what is happening when Excel gets slow? What stories should we be telling people to help them make sense of all that? Latency, bandwidth, storage capacity, and malware are examples of computing realities that are hard (maybe impossible) to mask.

My colleague, Keith Edwards, does research at this intersection of HCI and Computer Science Education Research, according to my definitions. He asks what he can do to help lay people make sense of their home networks and of the security risks that face them. He addresses these questions as an HCI designer, coming up with new metaphors to explain networking and security and creating interfaces that support those metaphors. He starts these projects by asking people how they think that their network works and what a security challenge is. Making sense of that data and figuring out how to improve that worldview is the domain of CS Ed Research.

The closer the user's tasks get to controlling the raw machine, the harder it is to come up with a layer that hides the computational details--and the further one is from the power of the raw machine. Alice is one of those layers. It's a great programming-for-storytelling world, but it limits the kind of programming that one can do. Can we make iteration or conditionals or sequential execution look like something else, something easier to understand and manipulate? How much power is given up as one accepts a layer on top? Does that tradeoff limit what people want to do with computing? We don't know much about what the tradeoffs are between layers and computational power.

To make computation accessible and powerful for users, we can change computation. That's the job of HCI. When we reach the limit of HCI (the HCI cliff?), we are in computer science education research. What are the users understanding, and how can we help them understand? At that limit, the game is just like psychophysics--we have a fixed reality, and now we can study how the nature of humans interacts with that reality.

To me, these are fascinating and important questions. Computing is the most important human invention since writing. How do we come to understand it? Why do people try to understand it? For what purposes are people interested in working off the edge of the HCI cliff, where the task is to wrestle with the computer itself? How does that motivation help people to overcome the complexity of programming? How do we make programming more approachable, more attractive, more worth wrestling with? What is it that's hard about programming? What do people think the computer is doing when they use it? Do they have a notion of how the computer works when they use applications like Excel? How do we measure what people know about computing?

Does answering these questions fall into a computer science department, into human-centered computing, or into interactive computing? It depends on how one defines these departments or programs. These questions should be explored. Maybe they should be explored in psychology, sociology, education, or even anthropology. Computer scientists are the ones who understand computing the best, and are thus the best situated to understand how humans are making sense of computing.