Two Themes: Language Bias and Informal Education on Secure Coding
Kathi Fisler, Associate Professor of Computer Science, WPI
A White Paper for the “Future Directions in CS Education” Summit

This white paper describes two separate research thrusts in computing education. One centers on the effects of linguistic bias in studies of how people learn to program; the other centers on secure-coding education for self-taught programmers. I am concerned that discussions of computing education are too dominated by questions about novices in their first computing course. The second of my thrusts looks at an important area of computing education that people begin to learn after having foundational background in programming, but that requires modes of thought beyond introductory programming.

Have language features biased foundational results on learning to program?

The Sapir–Whorf hypothesis suggests that language both determines thought and affects the cognitive structures through which people understand concepts. While this hypothesis was framed relative to natural or human language, it also raises interesting questions for computer science.

Most of the classical papers of computing education worked from imperative, control-dominated programming languages (such as Pascal). The programs studied in these papers are dominated by control structures: various forms of loops, conditionals, procedures, and I/O, with arrays as the primary (if any) data structure. The observations in these papers identify challenges in learning to write, understand, and debug programs. Much ongoing research builds on these observations. Unfortunately, relatively little work has explored the extent to which the observations in these papers are an artifact of imperative programming, and not endemic to programming itself.

Consider Soloway’s classic “rainfall” problem, which asks students to read in numbers until a sentinel is entered, then produce the average of the non-negative inputs (without the sentinel). Soloway used this problem to explore how novice programmers compose goals and plans: a standard solution would use a single control loop to weave together plans for consuming inputs until the sentinel appeared, ignoring negative numbers, counting inputs, accumulating the sum of inputs, and guarding against division by zero. The challenges include knowing which loop to use and knowing to compute the count of inputs within the loop but the average outside of the loop. Students have historically performed quite poorly on this problem, across several replication studies, even when given the sequence of inputs in an array.

A student with lists at their disposal, however, could approach the problem quite differently. The student could first read all of the inputs before the sentinel into a list, then write separate functions to remove the negative numbers, traverse the list to count the elements, and traverse the list to sum the elements before producing the average. This plan composition is vastly easier (sequential, rather than woven together). It requires students to be comfortable using data structures and writing lots of small functions to perform individual parts of tasks. Both traits are common in scripting and functional languages. Might the rainfall problem then be easier for students first learning such languages?
As another example, consider Pea’s “superbug” observation, which proposes that students struggle with programming because they initially interpret constructs through the semantics of similar terminology in English. Pea argues that using English interpretations of terms like “if” and “while” without could explain many errors that students make in novice programming studies (particularly in the context of sequential statement execution). Functional programming, in contrast, uses fewer terms and structures that readily map to English. Instead, functional programming more closely aligns to algebra as taught in middle- and high-school (in the United States). It seems unlikely that Pea’s observed “superbug” would arise in an introductory course based on functional programming.

These two examples are not arguing that switching paradigms would remove all challenges of learning to program. The challenges would be different, likely easier some ways and harder in others. The point is that we have generalized some observations about learning to program from a narrow space of languages. Courses that use different languages teach students vastly different material in rather different ways. These differences are particularly evident when one looks at attempts to develop cross-language “concept inventories” for CS1. As someone who has long taught introductory programming through functional languages, I am struck that my students are rarely prepared to answer a single question on these exams after a semester of programming. Yet they have covered a rich collection of computing content—including data structures, recursion, functions, and abstraction—that are also fundamental to computing education.

Understanding how programming knowledge develops in different paradigms will help the CS Education community make better recommendations for the different contexts in which we now expose people to computing. The 2013 ACM Curriculum Committee embraced paradigm-neutrality in CS1 (moving most paradigm-specific concepts to the Programming Languages knowledge area, and out of the now-defunct Programming Fundamentals knowledge area). The CS education community now owes it to our colleagues to consider our foundational results across linguistic paradigms.

In considering the impact of paradigm on computing education, we should be sure to explore impacts on teachers moving into computing as well as on students. The CS10K effort has put significant emphasis on recruiting new teachers to CS Principles. Some teachers will migrate into teaching computing more gradually, through teachers who bring computing modules into their non-computing classes (in math, science, arts, etc). Each of these teachers starts from significant disciplinary knowledge, some of which will resonate better with some approaches to computing than with others. Functional programming, for example, stands to be much easier for math teachers given their familiarity with algebra.

The following problems frame a research program on linguistic diversity in learning to program:

- Replicate the classic studies of computing with students whose first programming course emphasizes (a) data structures that are traversed more easily than arrays, (b) small scripts or functions, or (c) expressions rather than statements. Courses based in Python, Ruby, Scheme, and Haskell are good candidates, in particular because the pedagogies underlying these languages can differ substantially from one another.
Identify the different challenges that arise in each combination of the features from the previous studies. Which are shared across paradigms? Which seem easier or harder for us to overcome?

Design studies to understand how students bring knowledge of high-school mathematics to bear on learning to program in a functional model. Experience suggests that students have fairly weak understanding of algebra from high-school. Does that weakness help or hurt when students learn to program effectively with functions?

Explore whether languages that are semantically closer to mathematics provide a more gradual path for in-service math teachers to begin teaching computing.

Explore whether the end of CS2 is a better point at which to expect a concept-inventory checkpoint to apply across approaches to introductory computing.

**Expected Impact:** Within five years, we could collect enough data on learning in different paradigms to develop an initial picture of which foundational results generalize and which do not. We could gather data on learning under different paradigms across different populations of learners (such as teachers, students at various levels, etc), including an initial picture of the different problem-solving strategies that emerge in each case. These results would help us identify finer-grained questions to compare learning effects across paradigms.

If we find that vexing problems in CS education manifest differently across paradigms, we would be in a position to better tailor our education to our diverse audiences. If we do not find differences, we will at least know to look beyond largely linguistic solutions to the challenges of early CS education.

**Secure- and privacy-aware-coding education for self-taught programmers**

Security and privacy are arguably among the biggest challenges facing modern computing. Extensive efforts are underway to teach people how to safeguard their privacy, identity, and data in our increasingly interconnected world (through the National Initiative on Cybersecurity Education). Smaller efforts aim at comprehensive security curricula for professionals in computing-related fields. As end-users and computing professionals are at two different ends of the security education spectrum, these trends suggest broad coverage of security education relevant to computer science.

At present, however, relatively less work targets a dangerous population in the middle of the spectrum: the lightly-trained or self-taught programmer with no background in security who provides a mobile application or web service to an unsuspecting public. Mobile app markets, particularly Android’s, provide little oversight on what people upload for distribution. Anyone can make a website; many small businesses and nonprofits have sites developed by people who just “figured out” a bit of web programming. Once those sites start gathering data from users, or mobile apps misuse granted access to user data, they can rapidly create security or privacy problems.

The democratic marketplace that allows anyone to distribute software for others contributes to the problem, but remains one of modern technology’s greatest strengths. As educators, we leverage this marketplace constantly to engage students: classes create mobile and web applications that students
can share with family, friends, and beyond. People can create interesting and useful software and services with relatively little training and some help from online forums; creating safe software is harder. A little knowledge can be a dangerous thing, particularly around security and privacy.

Computing education thus faces the challenge of raising the bar on secure and privacy-aware programming, particularly for people who take at most one or two formal computing classes. The “natural” suggestion that secure programming be part of CS1 is misguided. One cannot think effectively about programming securely before one thinks effectively about programming in general. CS1 rightly focuses on understanding small programs; many security problems only arise when small programs are part of larger systems. While some common security holes could be discussed in CS1 (such as input validation), it is not clear that we should ask students to properly handle malformed inputs before they can handle well-formed ones. Furthermore, CS1 rarely teaches the programming languages used to write real-world web and mobile applications (for good reason!). The best we might hope for is that CS1 raises student awareness that software can create security and privacy problems; we cannot expect CS1 to teach the solutions, or even deep understanding of the problems.

If security does not belong in the small number of formal courses that introduce self-taught software developers to programming, how do we engage them in this area once they are learning and coding on their own? This question depends heavily on CS education research, because we do not currently have a clear sense of when people are conceptually ready to appreciate essential security concepts. Many security professionals refer to a “security mindset” that encourages people to see potential flaws and weaknesses in system designs. Many debate whether this mindset can be—or needs to be—taught. This white paper presumes that security mindset can be taught (at least to the extent needed by developers of non-critical software). As such, it focuses on the educational research needed to understand how people learn to think about software system failures and vulnerabilities.

Advancing informal secure programming education would benefit from answers to several questions:

- What are the conceptual pre-requisites for various best practices in secure software development? How much prior computing background does one need to understand and apply various best practices?
- “Security cannot be bolted on” is the mantra of secure-system design. The justification for this mantra is hard to appreciate on small programs, but late-novice programmers work primarily with small programs. Are there effective ways to teach this concept before someone is able to write large programs?
- Does introducing secure programming through intra-procedural concerns (such as input validation) on small programs create an inaccurate initial perception that security can be bolted on? If so, does this create false confidence in one’s secure programming skills?
- Security is related to testing, in that each asks developers to think about system behavior under various inputs. Does an emphasis on testing in early formal computing classes help a student develop security mindset more readily outside of formal classes?
- Using games to teach concepts can be problematic if students focus on the process and mechanics of the game itself, rather than the material the game strives to teach. Security is
often viewed as a game between a developer and an adversary, which creates opportunities to align gameplay processes with the material we need to teach. What conceptual prerequisites regarding programming do people need to appreciate the game model of security? What kinds of game playing help people transfer intuitions about adversaries to thinking about secure software design?

- How does “security mindset” develop in programmers? What experiences distinguish developers who take security seriously from those who do not? How do we create effective awareness-building experiences for late-novice programmers?

**Expected Impact:** To the best of my knowledge, little CS education research is looking at the cognitive foundations required to understand secure software design. General research on learning about systems and failure exists, but computer scientists have not explored how that work applies to the particular system models that underlie web applications and mobile applications. This context matters, because (a) self-taught programmers who want to create these applications are often not learning in the general setting of CS courses and (b) CS Ed research has been demonstrating the importance of context in computing education.

This research program has obvious impacts on creating effective educational materials around security and privacy. Long-term, its more significant impact should be on user-centric software-development tools. If computer scientists better understood programmers’ mental models of security, we could create programming tools that help avoid certain security flaws. Such tools are particularly important when they offset programmers’ lack of training in advanced computing concepts.

Within five years, we could (a) develop hypotheses about the conceptual prerequisites for various security best-practices, (b) conduct exploratory studies about student readiness for various security concepts at different stages of formal CS education, and (c) explore the mental models of security and privacy help by casual app developers. These would provide the formative information required to explore ways to effectively teach our target material.

One might question whether the group of self-educated app developers is sufficiently large to warrant federal research investment. This audience is admittedly small, with large populations outside of the USA. However, if our existing efforts to broadly expand computing education succeed, the number of people trying to use computing with limited training will also grow. In this sense, this research program tries to get us “ahead of the 8-ball” around casual computing with sensitive data.