TEACHING STATEMENT

Kristopher Micinski (kris@cs.haverford.edu)

Passion for teaching motivates every aspect of my scholarship. I taught as often as possible in graduate school. Upon graduation, I further committed to developing my teaching by taking a visiting position at Haverford. To hone my teaching, I practice a lot. I write blog articles, and I make videos to demonstrate concepts in ways I don't cover in class. I work hard to make myself available to students.

First Steps: Teaching Assistantships I served as a TA for two courses at Maryland. The first was an undergraduate programming languages course. I taught discussion sections and led office hours. It was here I first discovered a wide variation in learning styles. Some students learn best with pictures. Others via example code. I continuously work to identify and account for these different learning styles.

The second course I TA'd was a senior-level compilers course. I developed new projects collaboratively with a faculty mentor. I learned to anticipate sources of confusion and imprecision and swiftly correct them. I wrote a key for each project, then systematically asked how a student might approach each decision I made differently. The goal of my projects is to build a mental model by tackling difficult questions, not to figure out the solution I came up with first. This is something I work to remember.

Teaching Undergraduate Programming Languages I taught my first course—Organization of Programming Languages—during the summer of 2015 at Maryland. My course had roughly 40 students.

I followed an active-learning approach and carefully constructed each lecture to focus on live-coding and group work to illustrate concepts. I began most classes by posing a question we could not yet solve. This helped motivate why we needed technical devices we would learn about during each class. I then gradually introduced new concepts, leading students to solve this problem. Each step along the way, I had students work together—frequently relying on interactive tools to check their work. For example, during our coverage of state machines I built a web app to construct and run finite automata. At the end of each class we had an artifact that students could use to begin a follow-on project.

Consider teaching regular expressions. I told students we would parse a gradebook file to calculate a class average. I began by showing students necessary boilerplate code to manipulate and read files. I find that this allows students to be motivated by real-world applications but avoids their getting stuck on minutiae. I then asked students to form groups and discuss a high level algorithm for calculating a mean. Engaging students early with simple questions helped everyone feel confident and motivated throughout class. I would then incrementally guide students toward a solution. For example, I would show components of regular expressions in an interactive tool then ask groups to recognize parts of the file. I've found that forcing students to stay checked in prevents them feeling overwhelmed.

I developed new projects to complement these exercises. For example, I had students implement an interpreter for core-ML. This project reiterated topics students learned in multiple ways. First, I believe the best way to truly learn a language is to implement it. Students had to precisely implement otherwise-hazy concepts like closures and call-by-value semantics. Second, students frequently have trouble relating math they learn in class to its implementation. Last, one role of mid-level courses is to write nontrivial amounts of code. This project addressed all three. This strategy of structuring projects to drill down into concepts after teaching them at a high level helps students internalize material.

Principles of Programming Languages at Haverford I am currently teaching this sophomore-level course on programming-language paradigms (assembly, object-oriented, functional). I've used this course to experiment with my pedagogy and reflect on teaching while a colleague (David Wonnacott) gives me frequent feedback.

I have already noticed and worked to correct several shortcomings, some of which were because of inexperience, others because I hadn't before taught at a small school. I noticed that I began the course with a tendency towards a deductive style, which was particularly challenging for students whose backgrounds on certain topics were shaky. Some of my students hadn't taken a computing course in a year, or took a their introductory sequence at a sister college. As a consequence, I switched to a more inductive style that focuses on review of small pieces before stating a concept in general terms.

As an example, I cover pointers in my course. I worried that students would be eager to use pointers and manual memory management to implement their projects. So instead of a full presentation where I cover the heap and heap-based allocation, I start with pointers to stack-based datastructures (we had previously covered the stack in assembly). I was careful to illustrate variable lifetime and then relate this to implementation in assembly. Only once students understand this simpler style do I then introduce heap-based memory allocation and management.

A key part of my philosophy is to *teach high-level concepts within a context to which students can directly relate*. I do this by timing introduction of technical material with projects that exercise those techniques. In a recent project students had to implement a dictionary that mapped strings to either functions or lists of strings, which wasn't easy to do using the types we'd learned so far. I introduced subclasses and developed in class an example that used subclasses to solve a reduced variant of the project. Students told me the timing was effective in helping them understand, and many successfully generalized the ideas to a solution.

Developing Computer Security I will teach a new course at Haverford starting in the Spring.

My course covers computer security from two parallel perspectives: attacks and defenses. Students love attacks. It gives them immediate feedback that they've understood something challenging. But when their systems get attacked they realize they need to learn defensive programming. We'll do both.

Most students won't regularly break into other systems. The reason I'm teaching this course is to give them good intuition for what it means to write correct code. At each stage of the course, we'll raise the meterstick a bit higher. For example, even if the code is memory safe, it might implicitly leak a secret key in its running time. students will learn that they need to continuously question what it means for their code to be correct.

Research as Teaching I currently advise six undergraduate theses. My goal is to teach students that "research" is more than reading papers and books, but confronting problems they can't yet articulate.

I have found that one particularly effective mechanism for undergraduate research is to have students apply an abstract concept using a large-scale research framework. For example, with one group of seniors this year, I am exploring how social-media permissions are used in Android apps. This is an underexplored area that will lead to a good publication once the work is done. But tackling this requires my students to glue together systems that don't quite do what they want, and I view this as a crucial experience.

Reflection and Growth I work for an active dialogue with a strong sense of trust, transparency, and fairness. I set up anonymous channels for students to provide feedback and let students know they are free to give criticism without damaging their grade. Students often take me up on this.

Last, I work to make myself highly available to students. I use online forums such as Piazza, and offer to Skype with my students. I take care to ensure this isn't a detriment: rather than give students answers I work to quickly identify where they're struggling. I then give them a concrete assignment and have them get back to me.