

My teaching is shaped by my research on program verification, where the central technical question (how do we communicate precisely about software?) turns out to be inseparable from the central pedagogical one. The proofs and specifications that verify a real system are dense and built on layers of abstraction that resist casual reading. Making such material accessible without diluting it is the problem my research has come to focus on, and it is the same problem I work on in the classroom.

Research-driven pedagogy

As my proofs about kernel code grew, the people I most wanted to discuss them with could not engage with the formalism alone. So I added representations that meet readers where they are: a page of separation logic and a hand-drawn diagram of the same invariant are not redundant, because each catches errors the other hides. This conviction became specNL, an open-source project that parses controlled-English specifications into checked separation-logic proofs; its tutorial site and editor support, which show a student the proof obligation their sentence produces, are teaching materials as much as research artifacts. When I introduce a formal concept, I begin with the informal description, in English, of the property a programmer would actually care about. Students write it in their own words; then I show them what the formalism captures that their description left ambiguous, and what their description captured that the formalism leaves implicit.

Approach to hard material

Three principles follow. First, every formal concept is taught with at least two representations: a natural-language paraphrase, and a diagram, a concrete example, or an execution trace. Second, students translate in both directions: English to logic catches ambiguities in their thinking, and logic back to English catches gaps in their understanding of the formalism. Third, every definition is followed in the same class by a non-example, an object that almost satisfies it but does not.

My PLOS 2025 paper, *Modal Verification Patterns for Systems Software*, treats verification as engineering with reusable patterns, giving students a recipe: name the resource context (an address space, a snapshot) and the steps between contexts, and the specification's shape follows. Students state it in English with specNL first, then prove it.

specNL also sets the ground rules for AI in my courses. In introductory courses, and wherever the meaning of a program matters, students may use an AI assistant to draft a specification, but specNL checks the draft first and automatically rejects any sentence that does not yield a well-formed proof obligation, before the student builds on it. Students then work out what the AI left vague.

Generative tools now produce plausible code on demand, but they cannot reliably explain why that code is correct or where it will break; the habits that protect against those failures (reading code carefully, articulating the invariant a piece of code maintains, sitting with a proof obligation until it makes sense) are what these principles are designed to build.

Experience

As a teaching assistant for CS 360 (Programming Languages) and SE 311 (Software Engineering) at Drexel University, and earlier for Software Engineering at Koç University, I led discussion sections for students with very different preparation: some had programmed since high school, others had come

to the major later. The students who struggled with type systems or operational semantics were rarely struggling with the symbols; the symbols had arrived before the questions they answer. Working backward from a confusing program to the property that would have prevented the confusion helped, and it helped the less-prepared students most. Pacing one section for that range is something I am still working on; my current approach is to make the multiple-representation structure explicit so that students can enter through whichever representation matches their background. My industry experience, as a senior software engineer at Crytek and as a research intern at Microsoft Research and BedRock Systems, also lets me ground software design and testing in the realities of large codebases.

Courses

I am prepared to teach across the core of a computer science curriculum: introductory programming, data structures, discrete mathematics, computer organization and architecture, and operating systems, as well as programming languages, compilers, the theory of computation, and software engineering. I would like to develop electives and graduate courses on the formal verification of systems software (separation logic, Iris, and the modal abstractions behind my research) and on verifying AI-generated code, and an operating systems course in which students build a small RISC-V kernel in Rust. In each, students first describe in English what a piece of software or hardware promises, then build and test it, and finally state the promise precisely and check it.

Mentoring and inclusion

For several years I have been a long-term mentor in SIGPLAN-M, ACM SIGPLAN's mentoring program, working with graduate and senior undergraduate students, many from groups underrepresented in programming languages research. Effective mentoring requires the same discipline I apply to formal material: meeting students at their current understanding, treating their confusions as information about how the material should be presented, and helping them build the scaffolding to keep learning on their own.

The same philosophy shapes how I advise research. In summer 2026, Taro Sekiyama and I advised Keisuke Nishimura, a research intern at the National Institute of Informatics, on verifying inline assembly. He had not met Iris or the construction of a program logic before. My principle is that a student should spend their effort on the problem, finding the right abstractions, rather than on the bits and intricacies of the implementation. So I took the implementation side off his shoulders. In our weekly meetings I explained the Iris concepts he needed by walking through existing developments for other logics in detail, showed him where to find what he was looking for, and answered his implementation questions as they came up. That freed him to concentrate on the design of the semantics and on what would make the result striking as programming languages research. Within the summer he mechanized the semantics and its first proofs himself. I advise students the same way: choose problems whose abstractions are worth a student's attention, give them the infrastructure so that the implementation does not swallow the idea, and meet every week.

I want students to leave my courses able to say precisely what software should do and to show that it does it, a skill that matters more as machines write more of the code.