David Van Horn

Teaching Statement

Dept. of Computer Science & UMIACS
University of Maryland
A.V. Williams Building
College Park, MD 20742

www.cs.umd.edu/~dvanhorn
dvanhorn@cs.umd.edu
(202) 460-4104

My approach to teaching computer science is based on the proposition that writing programs is the most precise form of thinking. As such, everybody should—and can—be taught to program as part of a college education. This approach permeates my teaching at all levels, from introductory programming courses to graduate level research courses.

I have taught courses at all levels. My graduate course *CMSC 631: Program Analysis & Understanding* introduces the complementary areas of *PL* and *program analysis* and exposes students to the basic principles of research processes in CS. It covers the theory and practice of PL and techniques to mechanically reason about programs and trains, while also training students to articulate questions and recognize elements of solutions.

At the upper undergraduate level, my course *CMSC 430: Introduction to Compilers* introduces students to one of the most powerful and fruitful ideas in computer science, which is that often the best way to solve a problem is to develop a new language that makes the solution easy to express correctly, succinctly, and maintain-ably. Student develop the skills needed to design and implement their own programming language. Throughout the course, students design and implement several related languages, and will explore parsing, syntax querying, data-flow analysis, compilation to byte-code, type systems, and language inter-operation.

At the introductory level, I designed and taught a two course sequence: *CMSC 131A, 132A: Systematic Program Design I & II*. The first course introduces computing and programming. Its major goal is to teach students principles of systematic problem solving through programming. It exposes students to the fundamental techniques of program design: an approach to the creation of software that relies on systematic thought, planning, and understanding from the very beginning, at every stage, and for every step. It uses interactive, distributed, multi-player games to engage students based on *Realm of Racket*, a book I co-authored with a group of undergraduates [1]. The second course studies class-based program design and abstractions for reusable software and libraries. It covers the principles of object orientation and examines the relationship between algorithms and data structures. Both courses help students develop critical thinking and general problem solving skills. They learn how to structure their ideas and articulate complex concepts to themselves and to machines. This sequence was offered on an experimental basis, working closely with the Associate Chair of Undergraduate Education to evaluate its success.

Diversity. I am deeply committed to increasing diversity in computer science. While representation of women and minorities have grown significantly in recent years, growth in both metrics is still needed. My educational initiatives, which target students in their first-year, are built upon methods that have proven effective in reaching under-represented groups [2], are designed to help recruit and retain students who traditionally have high attrition rates and low representation within CS programs.

I am currently serving in my second year as the Chair of the SIGPLAN Programming Languages Mentoring Workshop at ICFP. The purpose of the workshop is to encourage undergraduate students, women, and underrepresented minorities to pursue and thrive in research careers both in programming languages specifically and in computer science generally. The workshop encourages graduate students (PhD and MSc) and senior undergraduate students to pursue careers in programming language research. It provides technical sessions on cutting-edge research in programming languages, and mentoring sessions on how to prepare for a research career. It brings together leaders in programming language research from academia and industry to give talks on their research areas. The workshop engages students in a process of imagining how they might contribute to our research community. Travel scholarships are supported with funding from corporate sponsors and the National Science Foundation, with priorty given to students from under-represented groups such as African-Americans, Alaskan Natives, Native Americans, Hispanic Americans, Native Pacific Islanders, and women.

I have graduated one female PhD student (Dionna Glaze, now at Google) and mentored a female postdoctoral fellow (Niki Vazou, now TT-faculty at IMDEA). I have worked with many female undergraduates, including co-authoring a book with three women from CS1 [1]. I have advised female REU students; and trained more than a dozen women as undergraduate tutors. At Northeastern, I trained many students to teach the Bootstrap curriculum in after-school programs in the Boston area. Most of these programs were offered in economically under-served, predominantly African-American and Latino middle-schools.

Mentoring accomplishments. I have had the privilege of working with an outstanding group of graduate and undergraduate students during my time as an Assistant Professor. I have served as thesis advisor for one graduated Ph.D. student, who is now an Assistant Professor at the University of Vermont. I have supervised two graduated M.S. students. I have supervised three undergraduate REU students and mentored several others. One of these students is now a Ph.D. student at University of California at San Diego; another is a Ph.D. student at University of Colorado at Boulder. I have two top-tier publications in programming language outlets with two UMD undergraduates and another second-tier publication with a third. During my time as an Assistant Professor, I have published a dozen papers, mostly in top-tier venues, featuring a junior UMD colleague (undergraduate, graduate, or post-doc) as first author. I have supervised five post-doctoral students and helped placed four of them in tenure-track faculty positions at the University of Texas at Dallas, the University of Alabama at Birmingham, the University of Colorado at Boulder, and the Madrid Institute of Advanced Studies (IMDEA), and one in research and engineering positions at Microsoft Research.

I have served a three year term on the steering committee for ICFP. I have served on the PC and ERC of 23 conferences and workshops, including three flagship SIGPLAN conferences (POPL (3), ICFP (3), and OOPSLA). I was the program and general chair for the Symposium on Trends in Functional Programming and for the Workshop on Higher-Order Program Analysis. I have co-organized international meetings at NII in Japan and the regional New England Programming Languages Symposium and served on two NSF review panels. I have served on the ACM SIGPLAN Student Research Competition committee at ICFP and PLDI and chaired the competition at ICFP. I have participated in PLMW (and currently chair it), OPLSS, the PLT Redex Summer School, and Student Research Competitions; given guest lectures in *CMSC 396H: Undergraduate Honors Seminar*. For the past four years, I have chaired or co-chaired the committee that organizes our annual assessment of graduate students. I've served multiple times on the committees for graduate admission, Middle States evaluation, education, and the Iribe Center currently under construction.

Course design. I have designed and taught courses at the introductory and senior undergraduate level and PhD level. Three courses (131A, 132A, and 631) have been designed from scratch; one (430) has been adapted from an existing course.

My most significant curricular design work has been the design of a new introductory programming course sequence. Central to the course is a concept called "the design recipe," which is a step-by-step prescription for computational problem solving. From an end-of-sequence survey, a student remarked:

> "I feel like the design recipe is immensely helpful in building and editing code, since it helps you visualize the beginning and end, helping you make a coherent body."

The steps of the process for designing a function, before attempting to implement it, include writing a succinct English description of the function's purpose: it's "purpose statement," and formulating input and output examples. Both guide the actual coding of the function, and the examples can later be used as test cases to confirm the correctness of the code. Another student wrote in the survey:

> "The design process was extremely helpful throughout both semesters. It's great that something as easy as writing down the intent of a method and examples of how it should work can be so beneficial. I expect the design process to help me in programming as long as I am programming."

In general, the central concern I have as a teacher is effectively developing skills and intellectual tools that students can rely on to help them approach problem solving. These explicit steps, once internalized by students, guide them

from problem analysis to solutions with well-defined intermediate artifacts at each step.

Another them to all my courses is an emphasis on the importance of collaboration in problem solving. This includes developing skills for communicating effectively, articulating design choices clearly, and working with others. As one of my intro students aptly put it, "coding is not solely about programming alone in a dark room, but instead is a collaborative effort by working together in pairs and groups."

CMSC 131A: Systematic Program Design I introduces computing and programming. Its major goal is to teach students principles of systematic problem solving through programming. It exposes students to the fundamental techniques of program design: an approach to the creation of software that relies on systematic thought, planning, and understanding from the very beginning, at every stage, and for every step. It uses interactive, distributed, multi-player games to engage students based on *Realm of Racket*, a book I co-authored with a group of undergraduates. It emphasizes use of specification, testing, and collaboration to build reliable software artifacts.

The courses help students develop critical thinking and general problem solving skills. Students learn how to structure their ideas and articulate complex concepts to themselves and to machines.

CMSC 132A: Systematic Program Design II builds upon the systematic problem solving foundation of CMSC 131A, covering class-based program design and abstractions for reusable software and libraries. It covers the principles of object orientation and examines the relationship between algorithms and data structures. It exposes students to industrial strength programming languages and tools and demonstrates how the problem solving and programming principles of the 131A carry over to new languages and environments.

Taken together, CMSC 131A and CMSC 132A provide an alternative path to prepare students for the rest of the Computer Science major, while simultaneously leveling the playing field for incoming students with varying levels of background in programming and relying only upon basic notions of arithmetic and algebra in order to convey programming and computational ideas.

CMSC 430: Introduction to Compilers introduces students to one of the most powerful and fruitful ideas in computer science, which is that often the best way to solve a problem is to develop a new language that makes the solution easy to express correctly, succinctly, and maintain-ably. Student develop the skills needed to design and implement their own programming language. Throughout the course, students design and implement several related languages, and will explore parsing, syntax querying, data-flow analysis, compilation to byte-code, type systems, and language inter-operation.

CMSC 631: Program Analysis & Understanding introduces the complementary areas of *PL* and *program analysis* and exposes students to the basic principles of research processes in CS. It covers the theory and practice of PL and techniques to mechanically reason about programs and trains, while also training students to articulate research questions and recognize elements of solutions. The course requires students to build precise, executable *models* of programming languages and semantic tools for reasoning about and analyzing software. The course is project-based, culminating in a capstone collaborative project that requires clear written and oral communication skills, sophisticated programming skills, and analytic problem solving skills need to overcome technical research problems.

References

[1] Matthias Felleisen, Barski M.D. Conrad, David Van Horn, and Eight Students of Northeastern University. *Realm of Racket: Learn to Program, One Game at a Time!* No Starch Press, San Francisco, CA, USA, 2013.

[2] Wendy S. McClanahan, Sarah K. Pepper, and Meridith Polin. "I program my own videogames": An evaluation of Bootstrap, 2016.