

Why Teach Programming Languages in this Day and Age and How to Go About it

Matthias Felleisen, PLT
Northeastern University,
Boston, MA

Abstract

The question is not whether a computing curriculum should include a rigorous course on programming languages, but which topics make up the minimum that we wish every student to understand, and how we should teach these topics.

Why Teach Programming Languages: Renaissance

For the past 40 years, people have repeatedly announced the death of programming languages. By the late 1990s, ACM's curriculum committee tried pushing it to the periphery of the standard curriculum.

Around the same time, the real world experienced a programming language renaissance. On one hand, Java and then C# brought research from the 1970s and 1980s into the mainstream. A look at the proceedings of our research conferences reveals how inspiring these events were. On the other hand, the regular world felt smothered by the evolving Java/Eclipse/Types monoculture and fled to dynamic scripting languages. Tcl/Tk, Perl, and Python took this world by storm; web browser introduced JavaScript, and AJAX made it indispensable. These languages are lightweight tools for customizing complex systems and libraries and for composing such products. When Greg Sullivan organized LL(1) in 2001, he expected a couple of dozen people yet 130 showed up; the excitement was palpable in the adjacent hallways. Northeastern's co-op department regularly reports a flood of co-op positions involving scripting languages.

Beyond mainstream heavy-iron languages and lightweight scripting languages, we also have a number of domain-specific languages. Database engineers formulate queries in them; data miners use them to describe the sea of bits; "aspects" started as a series of such special languages.

In short, “programming languages” as an area remains vibrant and central to computer science. Conversely, the quick pace of changes in the real world demand that we graduate students who understand the principles of programming languages (POPLs); who can use them to navigate the diverse and formidable maze of programming languages; and students who are prepared to design their own special-purpose languages *when* needed.

Why Teach Programming Languages: Constraints

Does “programming languages” matter as much as algorithms, complexity, data bases, ethics, formal languages, graphics, human-computer interfaces, multi-media, parallel programming, or operating and networking systems? After all, we can’t teach it all!

From any perspective we take, the answer is an emphatic yes. Whether we are teaching at a liberal arts college, a technical institute, or a university, our students must study how to design programs and beyond that, they must experience a course in each of the three intellectual pillars of our discipline: algorithms, POPLs, and systems. Everything beyond these foundations is luxury. Naturally, all of us enjoy luxury goods, and our students will enjoy the luxury of courses on application areas. If we fail to create a proper foundation, however, our curriculum will prepare them neither for a long-lasting career in computing nor a course of graduate studies. Put differently, the real question for most departments is not whether to teach programming languages but how much of programming languages to fit into an already over-crowded curriculum.

How to Teach Programming Languages: Principles

If we agree on a single, required course on programming languages, the question is reduced to the questions of *what* to teach and *how* to teach “it.” As for the content, three obvious topics emerge immediately:

1. syntax and abstract syntax, basic static properties (“linguistics”);
2. semantics, “dynamic typing,” and run-time libraries (“truth”);
3. plus type systems, soundness and unsoundness (“proof”).

Secondary topics depend on how we end up teaching the material.

Traditionally, programming language courses come in three different flavors: surveys of classes of languages (e.g., Pratt, *Programming Languages*); implementation-based surveys of linguistic mechanisms (Friedman

and Wand, *Essentials of Programming Languages/3e*); and mathematical approaches to semantics, type theory, and related topics (Mitchell, *Concepts in Programming Languages*). While many researchers may gravitate toward the latter, it cannot possibly serve the majority of students at the majority of colleges. I insist, though, that without some rigor, we cannot teach what “semantics” and “types” really mean. I therefore propose that

an interpreter-based approach should be the natural compromise for a course on POPLs.

While this course continues training students in program design, it can also cover the interpretation of standard linguistic mechanisms, including run-time checking; the design and implementation of type checkers; the meaning of types; the connection to run-time libraries; and ideally the workings of a garbage collector. Optionally, this course may include material on some formal language ideas (the power of regular expressions *vs.* context-free grammars) as well as the principles of compilation (separating static from dynamic aspects in a program), especially if courses on these topics aren’t in the standard curriculum.

Unfortunately, an interpreter-based course easily overwhelms students. Instead of connecting the course material and the real world, instructors and students often spend almost all their energy on the implementation effort. I therefore propose the following amendment to my original thesis:

the course must illustrate each covered linguistic mechanism with several examples from existing languages, ideally complemented by programming exercises (using the mechanisms or languages).

The syntax segment may include discussions on free-flow notation, white-space-and-indentation-included variants, and the power of parenthetical languages. When covering semantics, students should study how related constructs in “real” languages fail to live up to basic quality standards. Last but not least, since the introduction of linguistic constructs tends to address an abstraction problem, i.e., the lack of a mechanism for creating “single points of control,” the course should demonstrate such problems with small programming exercises in (selected real-world or constructed) languages.

Without a course that explicitly relates “our” ideas of programming languages to “their” reality, people will continue to call our area “dead,” a peripheral topic for the computing curriculum, and irrelevant to the world.