Why Teach Programming Languages

Olin Shivers

College of Computer and Information Science
Northeastern University
shivers@ccs.neu.edu

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Languages

Keywords Programming languages curriculum

Learning from History (and Physics, and Literature…)

In 1991, the year I received my doctorate, I was discussing my thesis topic with the dean of a top-three research department in Computer Science. He remarked, "I don't see what the issue is. The programming-language problem has been solved; the name of the solution is C."

Several years later, I reminded him of this claim, and, suitably chastened, he amended his point of view, saying, "OK, Java." So it does not surprise me that the importance of programming languages, as a field of study within Computer Science, is not understood as well as it should be.

My belief, on which I'll expand in this essay (or, more accurately, this collection of mini-essays), is that the study of programming languages (PL) is central to the study of Computer Science. I would go further, in fact: not only is PL a core component in the curriculum for students specialising in Computer Science, it is an important element of a general, liberal-arts education.

Consider other major elements of a liberal-arts education. We do not think it's important for college students to study Physics or History because they plan to become physicists or historians. We wish students to have knowledge of Physics, because it *illuminates the world in which we live and informs our experience in that world*—when we skid a car, for example, or pick up a heavy box, or vote on government proposals to fund ethanol fuel production.

We could say the same of History, or Literature, or Economics…or, for that matter, Computer Science: it is important for students to understand something about computation, as this also illuminates the world in which they live, a world whose connective tissue is increasingly composed of pervasive computational systems.

The mechanisms of programming *languages*, in turn, inform a student's understanding of computation itself, its power and its limitations. Studying these mechanisms, and—critically—putting them to work to design and implement programs, is the fundamental means by which students grapple with the nature of computation. The only way to really understand computation is to write programs; students can only write programs by means of programming languages.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

2008 SIGPLAN Workshop on Programming Language Curriculum , May 29–30, 2008, Cambridge, MA, USA.

Copyright © 2008 ACM [to be supplied]. . . $5.00

Languages and models of computation

Novice programmers, I have noticed, tend to be obsessed with details of syntax: e.g., are program blocks delimited by curly braces or `BEGIN/END` pairs? As students become more sophisticated, they focus more properly on the semantic elements of the languages they study and use: dynamic versus static types, class versus module structure, and so forth. Eventually, they come to see the core structure that distinguishes different programming languages from one another: behind every interesting programming language is an interesting model of computation.

- SQL, for example, expresses the model of the relational calculus: tables and joins and so forth;
- regular expressions capture the model of finite-state automata;
- Scheme is syntax for the untyped lambda calculus;
- Java and its OO relatives capture a communications-oriented model of computation where the fundamental computational elements are stateful agents that compute by sending one another messages;
- APL is a world of uniformly applying operations to regular, multi-dimensional arrays of data;
- . . . and so forth.

To a great degree, then, the study of programming *languages* is where the student is exposed to the full spectrum of *computational models* that have been devised by the mind of Man. Students who do not possess these models are intellectually impoverished; they do not understand computation.

Languages, and their associated models, shape the way we think about problems. One example I use to make this point to my students is to point out how expert Perl programmers, who do not possess a sufficiently deep understanding of computation, will frequently attempt to parse HTML text using the mechanism that Perl makes available with very low syntactic overhead: regular expressions. The idea of parsing a context-free language like HTML with a regular expression becomes my canonical example of doing the wrong thing due to intellectual provincialism: even smart programmers can't get it right if their horizons are too limited. They wind up trying to "drive a nail with a screwdriver."

Connecting computation to people

The theory of computability is a subject that encompasses the boundaries of all possible computations. Programming languages, in contrast, brings our focus in to describing the set of computations *constructed by humans*.

I tell my students, when I teach programming languages, that so-called "computer languages" are not actually for computers. Computers "understand" (that is, execute) binary machine code, not Java or SML. Computer languages (e.g., Java and SML) are actually for *people*: notations designed to be employed by humans to describe computations. The particular mechanisms of programming languages are designed to help humans think about the *design* and *structure* of computation: modularity, abstraction, scope, recursion, and so forth. These are human-centric mechanisms, created to help humans manage their cognitive deficits and leverage their cognitive strengths, in order to grapple with the construction of artifacts whose boundaries are typically determined not by clock speed or disk capacity, but simply by human limits on our ability to manage complexity.

Thus, the study of programming languages is essentially where the study of computation connects to people. PL is how we access the heart of the field.

Fundamentals and fads

I once read an article by a senior academic stating that "nothing changes faster than Computer Science; our curriculum should therefore change with equivalent speed." My immediate reaction was that "things that change rapidly" is just a long-winded way to say "ephemera:" that which is not of lasting value. Instead of

chasing fads, we should focus on *essentials*—core fundamentals that retain value across shifts in technology and fashion.

Deep structure is what has lasting value. Fortunately, PL has matured to a point where this structure can be articulated and taught. There are several excellent textbooks now available that are able to dissect programming languages into their component semantic elements, providing students with the semantic vocabulary needed to understand a wide spectrum of programming languages. Students must be equipped with this deep structure if they are to cross these shifts in fashion and ephemera successfully.

- By the time a student is in the junior year, a teacher should be able to begin a course with the statement, "We will be using Python (or C, or MIPS assembler, or SML, or Matlab, or some other language) for the work in this class. If you don't know it, go learn it." Coping with a new language encountered this way should cost a student no more than a week.
- Beyond the classroom, it is a critical requirement of people who produce software that they be able to acquire new languages and new language paradigms. (And the fraction of our workforce who will, in some form or another, need to program a computation at some point in their career will trend towards 1.0 over the next 25 years.) Someone who encounters, say, Ruby on the job needs the conceptual framework to say: "Oh, I see. It's basically a single-inheritance, class-based object-oriented language," and then rapidly and fluidly pass to the syntactic and semantic small details for this platform.

The problem with chasing fads is that when a new fad comes along five years after a student has left the university, we won't have access to the student to help him get past the surface novelty of the new thing. Only by arming this year's student with core intellectual tools—which certainly includes programming languages—will next year's ex-student be able to cope with rapid change.

I'll close by illustrating this claim by means of an example that is completely removed from my own personal linguistic enthusiasms. In the 1990's, students at MIT were taught the fundamentals of computation using Scheme. After this introduction, they moved on to studying the task of constructing complex, scalable software artifacts using a *horrendously* un-trendy programming language: Clu. Clu is a language that was born obsolete. It never achieved any significant industry penetration, to the slightest degree. But it was a language that cleanly and clearly explicated the mechanisms needed to construct complex software: exceptions, module systems, static types, objects and so forth. These structures, clearly presented, are what is important; once mastered, they can be transferred to other languages with the same elements.

Students absolutely managed this transfer. I learned about the impact of teaching undergraduates Clu not from the professors who taught using it, but from a student who took their classes. This student subsequently became the first student of mine who managed to amass a personal fortune in excess of twenty-five million dollars. By his own description, he did so by applying the intellectual tools he'd learned from the study of language paradigms—among them, Clu.

And he put them to work to construct a Scheme interpreter, which sat at the heart of his company's core product.