Teaching Programming Languages in a Post-Linnaean Age

Shriram Krishnamurthi

Brown University
sk@cs.brown.edu

Abstract

Programming language "paradigms" are a moribund and tedious legacy of a bygone age. Modern language designers pay them no respect, so why do our courses slavishly adhere to them? This paper argues that we should abandon this method of teaching languages, offers an alternative, reconciles an important split in programming language education, and describes a textbook that explores these matters.

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

General Terms Languages

Keywords Programming languages curriculum

1. The Paradigm Shift is a Shift to No Paradigms

The 1990s witnessed a Cambrian explosion in the family of programming languages. Whereas the decade dawned to a stultifying conformity, by decade's end two forces had reshaped the linguistic landscape: scripting and the Web. Scripting observed that a substantial amount of programming now focused on connecting libraries and utilities rather than creating them. The Web's standardized, lightweight interface meant that any language could lurk behind a server. (*On the Internet, nobody knows you're a Scheme program.*) Because many Web programs did what scripting languages supported best, the Web elevated these languages from hobbyist projects to entities enjoying widespread corporate support.

The teaching of programming languages (PL) has not kept pace with these developments. Many books have de rigeur chapters on scripting and on Web programming, but with chilling unsophistication. PL research has unearthed numerous lessons about the nature of Web, reactive, interactive, and asynchronous programming, but these have not made it into virtually any textbook.

Even more striking is what scripting languages say about the organization of languages. Most books rigorously adhere to the sacred division of languages into "functional", "imperative", "object-oriented", and "logic" camps. I conjecture that this desire for taxonomy is an artifact of our science-envy from the early days of our discipline: a misguided attempt to follow the *practice* of science rather than its *spirit*.

We are, however, a science of the artificial. What else to make of a language like Python, Ruby, or Perl? Their designers have no patience for the niceties of these Linnaean hierarchies; they borrow features as they wish, creating melanges that utterly defy characterization. How do we teach PL in this post-Linnaean era?

For several years I have been working on *Programming Languages: Application and Interpretation* (PLAI), which is my answer to this question. This brief essay lays out some of the vision behind the book.

2. Students as Language Designers

Computer science enables creation without respect for institutions, and scripting languages are one of its best and worst manifestations of this power. I tell students that many of them will go on to design programming languages: not large, industrial languages like Java, but small, scripting or domain-specific languages that will enhance their productivity. A good example is XACML, a language for separating the access-control logic from the rest of the program, using rules with domain-specific combinators. These students must therefore learn: (a) to recognize a scope for a domain-specific language, (b) the building-blocks that go into most languages, and (c) to avoid the mistakes of past language designers. This vision guides the book and my course.

3. Teaching the Other 90%

The textbook realm seems to be split between those that are rich in mathematical rigor but low on accessibility, and those high on accessibility but lacking rigor (and, often, even wrong). This puts professors in an unfortunate bind. I believe programming language design should be viewed as a *popular* activity, in the sense of one that every student must study. This is not because I believe every student *should* design languages, but because any student *might*. The student driven away by excessive mathematical rigor is, sadly, not necessarily going to refrain from creating his own language and inflicting it on the world. The tone, content, and style of PLAI is therefore designed with "the other 90%" in mind: the 90% of students who will never take an advanced languages class, and yet a few of whom may go on to create languages.

This also has implications for content. For instance, the majority of students have their heads filled with half-truths and falsehoods about garbage collection. In many departments, the PL course is the only place to correct these. That means the course should confront these misconceptions by comparing garbage collection to manual memory management. I also find that having students implement a collector or two makes an enormous difference, once they see that a mysterious procedure is actually just a relatively simple algorithm (conceptually).

4. Languages as Aggregations of Features

If languages are not defined by taxonomies, how are they constructed? They are aggregations of features. Rather than study extant languages as a whole, which conflates the essential with the accidental, it is more instructive to decompose them into constituent features, which in turn can be studied individually. The student then has a toolkit of features that they can re-compose per their needs.

This vision of systems as compositions of features is widespread in software engineering, especially in domains like telecoms. It is a natural vision to apply to languages, especially in constrained domains, where designers must wed the demands of the domain to general-purpose notions (abstraction, iteration, etc.).

As a design guideline, throughout the semester students re-examine the implications of the Scheme report's dictum, "Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary." By the end, I like to think, they have a much better sense of the mistakes common scripting languages have made, and why understanding and composing principled building blocks makes much more sense.

Of course, combining features demands also reasoning about their interactions. PLAI has some exercises about this, and my course's final project forces students to combine several features and understand the consequences. There is, however, still much to do in this regard.

5. Language Surveys or Interpreters?

Now for the continental divide of PL texts: a survey-of-languages or definitional interpreters? PLAI takes the position that this question represents a conflict without a cause[1] and interleaves the two approaches.

---

[1] With apologies to Beppe Castagna.

The survey approach has several benefits. By using multiple languages, students are forced out of today's Java monoculture. (Happily, every year a handful of students are turned on by Haskell or Prolog.) By writing (small) applications, they get a feel for how a distinctive feature (such as laziness or backtracking) can help and hurt. Most of all, they learn *why* one would want to study these languages in greater depth. On the other hand, they develop only a superficial understanding of these features, and may never get past a few examples to understand what something really *is*. In addition, they develop no skill at implementing even prototypes of languages.

The interpreter approach is essentially the dual. Most crucially, students learn what the features *mean*, but may never appreciate their implications. It can be fascinating to reduce the difference between eager and lazy evaluation to a half-dozen lines of interpreter, but does the student understand the enormous *consequences* of this difference?

These trade-offs are unsurprising: they are simply special cases of, respectively, inductive and deductive learning. The educational literature on learning styles tells us we should use both and, furthermore, that most students (remember the 90%?) prefer to proceed inductively. This, therefore, is what PLAI does. Each feature's definitional interpreter is preceded by programming activities that exercise that feature in the context of a language. Not only does this greatly motivate students, I argue that it helps them write their interpreter better because they already understand the desired input-output behavior. Most of all, they understand the software engineering consequences of linguistic choices.

6. Where's the Rigor?

One reader of a preliminary version of this essay worried that this approach to studying languages would not be "rigorous". Indeed, a lack of rigor is often a (justified) criticism of the survey approach. This is where interpreters come into play. Interpreters are simply stylized semantics, but by virtue of being programs, they have two advantages: (1) they are accessible to a broader range of students, and (2) they offer students a potent implementation tool for prototyping their own languages.

7. Innovations

PLAI has several innovations over most other PL textbooks. For instance, it discusses type soundness, a topic neglected in coverage of type systems, and type inference. It covers continuations through the agency of Web programming, an approach so accessible that only the weakest students now fail to understand the topic. Students understand garbage collection by implementing a pair of collectors using a high-level interface. The book increasingly features material on reactive programming languages. And so on; see it for yourself here (for free—like beer):

http://www.plai.org/

8. Specialized Languages

Some instructors have created specialized languages to partner the use of PLAI. Greg Cooper (now with Arjun Guha) created an excellent pair of languages for teaching garbage collection. Matthew Flatt very creatively built a dynamically-scoped Scheme, so students can immediately see the consequences of that scoping decision. Eli Barzilay has built a lazy Scheme. While I appreciate some of these less than others, I'm pleased to see this ecosystem flourish.

Acknowledgment