

Marketing the Programming Languages Course

Stuart Reges

Computer Science & Engineering
University of Washington
Seattle, WA 98195
reges@cs.washington.edu

Abstract

Programming languages as a required course is disappearing from undergraduate computer science programs. This is not surprising given that the course often proves to be challenging for faculty to teach and unpopular among students. The author argues that the best way to convince departments to retain this material is to emphasize the benefit to undergraduates of stretching their understanding of programming early in their careers.

1. Background

In the ACM's Curriculum 78 [1] report, "Organization of Programming Languages" was a required course. As "CS 8" it was one of only eight courses that the committee recommended be required of all undergraduate computer science majors. The history of programming languages as a required subject of study has been all downhill since.

In response to Curriculum 78, most departments created their own version of the programming languages course. In 1992, K. N. King described how the course had evolved since then [3]. He describes the course as "one of the most flexible in the computer science curriculum," noting that, "Different instructors cover different languages, different topics, and use different course organizations." This was a polite way of saying that the computer science education community could not agree on what this course was all about. King goes on to predict that, "With the publication of *Computing Curricula 1991*, the days of CS 8 are numbered."

Here is just a partial list of some of the challenges departments have faced in offering a required programming languages course:

- Ambiguity over course goals, content, and approach
- Textbook issues (this is the only course that tends to require multiple books—a central textbook and/or several language manuals, none of which is really what the instructor wants or what the students need)
- Staffing difficulty (a required course needs significant staffing, yet fewer faculty are familiar with the range of languages and topics typically included in the course)
- Inconsistency over time (each new instructor picks his or her own favorite languages/tools/concepts/approach)
- Producing good assignments (instructors often underestimate how difficult some problem will be to solve in a given language or how much students will struggle to learn a new language and environment)
- Lab issues (faculty and lab staff often struggle to provide a simple and reliable infrastructure for one programming lan-

guage, so when they are asked to support multiple languages for just one course, they understandably make it a low priority)

Perhaps not surprisingly, when the ACM unveiled *Curriculum 2001* [2], it included only 21 lecture hours of material on programming languages in the core and most of that material (e.g., 10 hours on "object-oriented programming") is covered by the introductory programming courses.

As a result, King's prediction has come true. Departments are increasingly dropping their required programming languages course as they respond to the pressure to cover an ever-increasing set of "fundamental" topics. Stanford, for example, has recently replaced their "Programming Paradigms" course with a systems course that includes coverage of concurrency and memory management. The memory management material isn't new, but it ended up there because it, in turn, had been forced out of the introductory sequence to make room for coverage of object oriented programming concepts.

I have personal experience with teaching the programming languages course at two different schools. This has given me an appreciation both of the challenges and of the opportunities involved in teaching this material.

2. My First Programming Languages Course

In the summer of 1998 my department head at the University of Arizona asked me if I'd be willing to teach the department's comparative programming languages course. I was surprised because I had listed it as my sixth choice. He pointed out that I was the only faculty member who had listed it as a choice at all and that the course had been taught by adjuncts in recent years. I wasn't sure how it would turn out, but I said yes.

The lack of interest among our faculty seemed unusual to me for a mid-sized department of eighteen that had been founded by a programming language designer. But as I talked to faculty and students, I began to understand the challenges involved in teaching this course well.

In talking to other faculty members, I found that:

- Faculty who had an interest in programming languages generally didn't have an interest in this class because it was too "nuts and bolts" for them. As one of my fellow faculty members put it, "Why would someone who loves music theory want to give people piano lessons?"
- Even faculty who were potentially interested were wary if the course involved using a language they weren't proficient in. Faculty don't like to appear ignorant in front of students, so all it takes is one language on the list of included languages to convince a wavering faculty member to say no.

- Even if you offer faculty the chance to replace a language with one they are more familiar with, they still hesitate because they know that they have to retool the course for this new language. Existing handouts, PowerPoint slides, and assignments won't be useful any more and might need to be replaced. Plus the instructor will have to pick an appropriate IDE and make sure that the lab is properly configured to support the new language. This is a substantial investment for a faculty member.

In talking to students, I found that:

- Students were frustrated because the course seemed to cover the wrong languages. They wanted to see programming languages that would get them a job, so they wanted to learn C++, PHP, JavaScript, or SQL. They didn't understand why a language like Prolog or ML would be included when nobody in industry seemed to care about those languages.
- Students felt they wasted a lot of time on logistical details. They didn't mind spending time understanding a difficult concept like how to apply map, but they didn't like it when they spent time trying to find a mismatched set of parentheses or trying to figure out how to load a saved program into a programming environment.
- Instructors for the course had often given out assignment specifications without solving the problems themselves. As a result, they often miscalculated about how difficult some task would be and the students ended up paying the price, which they didn't appreciate.

In putting together my version of the course, I decided that it had to be more about an experience than anything else. My first focus, then, was on better explaining the purpose of the course to the students.

I told my students that the purpose of the course was to stretch their understanding of what programming means. Taking them outside their comfort zone was not only okay, it was essential. I told them to think of the course as their foreign language requirement or their nonwestern culture requirement.

Looked at in this light, programming languages like C++ just aren't unusual enough to stretch their understanding of programming. Sure we could take a trip to New York City and we'd find it's different, but we can learn a lot more by traveling to Tokyo or Beijing. And who cares about what's currently popular? How much do you learn about music by studying popular groups like the Backstreet Boys or Britney Spears? You learn a lot more by studying lesser-known but more influential artists like Jimi Hendrix and Bob Dylan.

These analogies seemed to resonate with the students because they would repeat them back to me on course evaluations and when they described the course to other students who hadn't yet taken it.

To address some of the other concerns, I made sure that I picked programming problems that really showed off the features of the language. I often supplied supporting code that handled low-level details (e.g., user interaction in Prolog). That way the students could concentrate on the code where the programming language really shined. I also spent considerable time making sure that I had a good programming environment for each language that I knew would function well. I found it helpful to use the exact same programming environment myself while I prepared homework assignments and also when I lectured. I think it is extremely helpful to spend considerable time programming in front of the students to help them to understand how to use the new language and the environment you've chosen for it. Plus,

most of the languages we studied had convenient REPL style interpreters that are very conducive to a lecture format.

The course that I developed ended up being one of the most popular core courses for the major. My approach worked at Arizona because the course was intended to be a fairly practical introduction to a variety of programming languages without a lot of coverage of implementation issues or the concepts underlying programming languages.

3. My Second Programming Languages Course

At the University of Washington I have had a second chance to experience teaching a programming languages course. The UW course was similar in character to the course I had taught before, although instead of including a logic programming language like Prolog, the course instead covered two functional languages (ML and Scheme) along with Smalltalk. The course also included more explicit coverage of programming language concepts, although that had happened fairly recently after it was taught a few times by a PL researcher.

In the UW course I have seen many of the same issues that I saw at Arizona. Because it is required of all undergraduates, the department is forced to offer it all three quarters of the year. As a result, staffing the course is often a problem. The course also suffered many of the problems I had identified at Arizona.

Several different instructors have taught the programming languages course while I have been at UW, but only three of us have managed to get a student evaluation score above 4.0 (on a 5-point scale) for "overall quality of the course." For each of us, we have taught a fairly personalized course (personalized to our own strengths and interests). Other instructors have struggled to teach the course effectively, even with shared resources like lecture slides and sample code discussed in lecture.

On the plus side, the UW course includes a mini-interpreter assignment in which students implement some subset of a language. I hadn't been able to fit this into my Arizona course given the languages we covered and the time constraints of the course. It works well at UW because we use ML to give students practice with functional programming in general and then switch to Scheme, which is well suited to the mini-interpreter assignment.

I am now a big fan of the mini-interpreter style assignment. It fits in perfectly with my theme of stretching young minds. There is an "Aha!" moment that most students experience in working on this assignment that is difficult to achieve otherwise unless you are willing to require that students take a compilers course.

4. Selling the Course

Many of the faculty that I talk to don't like to think about the idea of marketing their course either to students or to other faculty members. But the truth is that if students don't care about what you're teaching them, then they won't remember any of it, and if fellow faculty members don't agree with your assessment of the importance of the course, then they'll vote to remove it from the list of required courses.

In dealing with students, I have found that the single most powerful argument for the programming languages course is the importance of the experience. "Look," I tell them, "you're going to have a career in computing that stretches over several decades. I don't know exactly what changes we'll see, but the one thing I can predict with certainty is that you'll see a lot of it. This field is very dynamic and you have to learn to anticipate and embrace change. The best way for me to prepare you is to give you experiences that stretch your understanding of programming. It's much better to do that now when you're just starting out. That

way when the changes come along, you'll be better prepared to handle them."

I used to make an argument like this in the 1980's when I taught students Smalltalk and ML. Now with a few gray hairs in my head I get to point out that my students in the 1980's complained that industry didn't care about Smalltalk and ML and yet by the 1990's, object-oriented programming had become the industry norm and companies like Sun were scrambling to figure out how to add the power of ML-like generics to Java and C#. My students also find it amusing to know that Guy Steele, one of the codesigners of Scheme, was also very influential in the design and evolution of Java.

When I talk to programming languages faculty, I often hear a list of concepts that students need to learn about. But when I talk to faculty outside the programming languages community, they don't seem to be very impressed with the idea that students should learn about closures or referential transparency or hygienic macros.

The argument that has been most powerful among computer science educators is that students must be exposed to different programming paradigms. For example, Dan McCracken, whose Fortran book was the first programming textbook I experienced, wrote in 1992 [4] that, "It is important that students understand several programming paradigms: imperative (procedural), functional, logic, concurrent, object-oriented, fetch/execute machine level. If students learn imperative only, with the others skimmed in an 'All Others' elective, they are stunted. They will apply the wrong tool in some cases, not knowing of others. They will be poorly prepared for a lifetime of learning, as our field continues to change."

The most convincing argument for continued coverage of programming languages material comes from this argument of preparing students for a lifetime of learning. We can try to argue this from the perspective of specific topics, but it becomes a more powerful argument when we talk about the experience rather than the specific topics.

I also think that we often don't do enough to increase the marketability of our own courses. The UW course I was asked to teach had been using Smalltalk. I asked why and was told that it was because it filled a niche as a dynamically typed purely OO language. Then why not Ruby? It might not be as clean of a language as Smalltalk and it doesn't have the same historical significance, but Ruby is wildly popular among professional

programmers and if it can get the job done, then why not use Ruby instead? I made this change when I taught programming languages at UW and it was probably the single most popular decision I made among the students.

5. Where Do We Go From Here?

As mentioned earlier, the programming languages course is disappearing from the list of required courses for undergraduate computer science majors. It might not be possible to prevent this from happening.

The best hope for preserving programming language material in the core requirements of computer science departments is to think carefully about the experiences that students should have and to articulate the importance of those experiences to our colleagues. As a simple example, everyone can see that concurrency is going to be an important concept for students to understand in the future. But if we limit it to just a topic to be covered somewhere, then it is just as easy to discuss it in a systems course, as Stanford has decided to do, as it is to discuss it in a programming languages course.

There is a strong case to be made that a programming languages approach to concurrency will be a more valuable experience to students because it will take place in a broader context of understanding the range of possible programming paradigms.

References

- [1] Richard H. Austing , Bruce H. Barnes , Della T. Bonnette , Gerald L. Engel , Gordon Stokes, Curriculum '78: recommendations for the undergraduate program in computer science—a report of the ACM curriculum committee on computer science, *Communications of the ACM*, v.22 n.3, p.147-166, March 1979
- [2] Computing Curricula 2001: Computer Science: Report of The Joint Task Force on Computing Curricula, IEEE Computer Society and The Association for Computing Machinery, December, 2001. <http://www.acm.org/education/curricula.html>
- [3] K. N. King, The evolution of the programming languages course, *ACM SIGCSE Bulletin*, v.24 n.1, p.213-219, March 1992
- [4] Daniel D. McCracken, Programming languages in the computer science curriculum, *Proceedings of the twenty-third SIGCSE technical symposium on Computer science education*, p.1-4, March 05-06, 1992, Kansas City, Missouri, United States