Injecting Programming Language Concepts Throughout the Curriculum: An Inclusive Strategy

Mark W. Bailey
Department of Computer Science
Hamilton College
mbailey@hamilton.edu

Abstract

As research in programming language design, implementation, and application advances, we must regularly revisit the undergraduate curriculum to ensure course content advances similarly. However, no matter how diligent our efforts, the undergraduate curriculum will continue to undergo pressure from all subfields of the discipline to include courses covering those subfields. Though we could advocate for core placement of programming language courses, it is likely, even inevitable, that a growing number of colleges and universities will choose not to mandate programming language courses for their degree programs [2, 5]. Thus, we must develop an inclusive strategy that encourages the teaching of programming language topics even in colleges and universities that choose not to devote an entire course to the study of them. I have proposed one such strategy that supports the injection of programming language concepts into other courses of interest to both students and faculty.

Categories and Subject Descriptors K.3.2 [**Computers and Education**]: Computer and Information Science Education—Computer science education; D.3.3 [**Programming Languages**]: Languages, Constructs, and Features.

General Terms Languages.

Keywords Programming language curriculum.

1 Introduction

Programming languages define the way we build, interact, control, and think about computer systems. They are fundamental to the way we teach computer science. Nothing demonstrates this better than our focus on teaching programming early—often before anything else—in our undergraduate curricula. Advances in computer technology have spurred enormous growth in both the discipline of computer science as a whole, and in the field of programming languages, in particular. The organizers of the Workshop on Undergraduate Programming Language Curricula posit that undergraduate curricula have not kept pace with this changing landscape. If true, correcting this problem is critical if we want our graduates to further advance the field of programming languages, the discipline, and become the most effective practitioners of the discipline.

2 The Real Problem: Increasing Pressure

Although this lag in curriculur development is undesirable, it has the potential to become much worse. Just as there have been advances in programming languages, there have been advances in many fields of computer science, as well as the birth of many new fields in the discipline. The result is the emergence of offerings, at the undergraduate level, of courses in bioinformatics, wireless networking, security, game programming, robotics, and mobile computing, to name a few. All of these offerings compete for a limited number of course slots in an undergraduate computer science major [3]. Only fundamental topics—as programming languages is—enjoy protection from these pressures in the form of "core" status in the curriculum. To prevent this core status from eroding, we must develop arguments and examples that articulate the importance of the field of programming languages and its instruction for all students. We already see this erosion where courses in compiler design, once considered core at most colleges and universities, have slipped to elective status, or are not offered at all.

Liberal arts colleges, in particular, have greater curricular pressures. Typically, these schools allocate only about ten courses for the major, reserving the remainder for breadth of study [4]. With so few courses, there is intense pressure to maximize the efficiency of each course in the curriculum. At the same time, there is pressure from students, parents, administrators, and industry to offer courses in the latest, "hottest," most lucrative technology areas. In such environments, it may not be feasible to dedicate an entire course to one field of the discipline, no matter how fundamental that field may be.

Given these curricular pressure realities, discussion of the role of programming language design, implementation, and application in undergraduate computer science education must include strategies to address the following problems:

- curricula may not reflect advances in the programming language field and technology,
- the core position of programming language courses will, inevitably, be questioned, and
- developing fields within the discipline will continue to exert pressure for inclusion in undergraduate curricula.

Though all of these merit further discussion, this paper focuses on strategies to support schools that cannot, or will not devote a core course to programming language design or implementation. I believe there are many, creative ways to inject the core principles of a modern programming languages course into the curriculum in other places.

3 An Inclusive Approach

Though enrollments in our compiler design courses have been on steady decline, we have had success teaching many of the fundamentals of compiler design in the context of a course that students perceive as being more relevant and appealing. I believe we can develop similar approaches for other topics within the programming language field. I base my approach on the observation that many topics in our field are so fundamental that they span fields and are applicable in a variety of course topics. For compiler design, we couch topics such as regular expressions, context-free grammars, the pumping lemma, the halting problem, program analysis, and program optimization in a computer security course on anti-virus techniques [1]. This results in a course that achieves our objectives of teaching many of the fundamentals of compiler design while simultaneously meeting students' expectations to study contemporary topics of relevance to today's computer professionals. The fruits of our efforts are increased interest and enrollments in the anti-virus course when compared to the traditional compiler design course.

We can use a similar approach to inject the principles of programming languages into emerging courses of interest to students and faculty. Every course depends, to some extent, on programming languages, language translation, run-time systems, memory management, virtual environments, or alternative models of computation. For example, a course in mobile computing could introduce the use of virtual machines, game programming often uses interpretation, and embedded systems require efficient memory management. By injecting these core principles into such courses, we will expose students to important concepts in the field, and we will reinforce the notion that these concepts are fundamental to the discipline, no matter what the context.

In order for this strategy to succeed, we must take the following steps.

1. identify the set of fundamental concepts in the programming language field,
2. identify potential target courses for injection of each concept,
3. develop supporting course material for each potential target, and
4. disseminate course materials.

Instructors will only adopt this approach if we can successfully illustrate, through mature course materials, that this approach enhances their courses.

4 Summary

As research in programming language design, implementation, and application advances, we must regularly revisit the undergraduate curriculum to ensure course content advances similarly. However, no matter how diligent our efforts, the undergraduate curriculum will continue to undergo pressure from all fields of the discipline to include courses covering those fields. Though we could advocate for core placement of programming language courses, it is likely, even inevitable, that a growing number of colleges and universities will choose not to mandate programming language courses for their degree programs. Thus, we must develop an inclusive strategy that encourages the teaching of programming language topics even in colleges and universities that choose not to devote an entire course to the study of them. I have proposed one such strategy that supports the injection of programming language principles into other courses of interest to both students and faculty. No matter what the strategy though, we must remember that our curricula must be as flexible as the languages we design.

References

[1] M. W. Bailey, C. L. Coleman, and J. W. Davidson. Defense against the dark arts. In *SIGCSE '08: Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*, pages 315–319, New York, NY, USA, Mar. 2008. ACM.

[2] M. Furst and R. A. DeMillo. Creating symphonic-thinking computer science graduates for an increasingly competitive global environment. White paper. Available at http://www.cc.gatech.edu/education/undergrad/bscs/threads_whitepaper.pdf.

[3] Joint Task Force on Computing Curricula. Computing curricula 2001. *Journal on Educational Resources in Computing*, 1(3es):1, 2001.

[4] Liberal Arts Computer Science Consortium. A 2007 model curriculum for a liberal arts degree in computer science. *Journal on Educational Resources in Computing*, 7(2):2, 2007.

[5] M. Sahami. Preview of the new undergraduate computer science curriculum. Slides, Apr. 2008. Available at http://cs.stanford.edu/degrees/undergrad/CurriculumRevision-Preview-04-03-08.pdf.