Some Thoughts on Teaching Programming and Programming Languages

John C. Reynolds
Carnegie Mellon University
john.reynolds@cs.cmu.edu

Abstract

It is argued that the teaching of programming is central to the education of skilled computer professionals, that the teaching of programming languages is central to the teaching of programming. that these topics must include the specification, structuring, and verification of software, and that they should be taught with the same regard to rigor and precision as in traditional mathematics.

Categories and Subject Descriptors K.3 [2]: Computer Science Education, Curriculum, also D.3 [0]

Keywords programming, programming languages, teaching, undergraduate curriculum

The question of what to teach about programming languages raises the prior question of what to teach about programming. In fact, the ability to construct reliable and efficient software is the central hallmark of the skilled computer professional, and the present-day rarity of this ability is a major cause of the sad prevalence of bug-ridden computer systems. Our educational institutions must help to solve this problem rather than to exacerbate it.

(Throughout these remarks, the terms "programming" and "programming language" will be interpreted broadly: By "programming" we mean the entire process of constructing programs, not just coding, and by "programming languages" we mean both executable languages and logics for specifying program behavior.)

Some argue that one can manage software production without the ability to program. This belief seems to arise from the mistaken view that software production is a form of manufacturing. But manufacturing is the repeated construction of identical objects, while software production is the construction of unique objects, i.e., the entire process is a form of design. As such it is closer to the production of a newpaper — so that a software manager who cannot program is akin to a managing editor who cannot write.

One might also argue that many computer professionals deal only with some specialization of programming appropriate to a particular application area. But these specializations vary widely and cannot provide the commonality that must underly a profession. It is precisely the general craft of programming that holds the profession together.

It is also the craft of programming that lies at the intellectual center of computer science (as central to that subject as classical mechanics is to physics), and has given rise to the concept of computational thinking.

Since programming languages are the languages in which we program, there is obviously a close connection between the teaching of programming and of programming languages. To teach one without the other is as implausible as teaching novel-writing without teaching natural language.

Most novelists, however, use a single language, while most computer professionals use many, quite different programming languages. Moreover, almost every programming language attempts to embody a particular style of programming, so that a broad education requires immersion in a variety of languages.

As Alan Perlis once remarked, "A good programming language is a universe for thinking about programming". In fact, it is entirely artificial and unproductive to separate the teaching of a programming language from the methodology it embodies. If one looks at particular aspects of programming languages, say type systems, or first-class continuations, or module declarations, it is clear that understanding will be impeded if the student learns to use these mechanisms separately from learning how they work.

In Perlis's sense, however, not all programming languages are good. Most — including the most widely used — have serious design defects, so that learning such languages is less a matter of mastering a style than of learning workarounds for the language designer's mistakes.

I believe that the most reasonable approach to this problem is to first learn to program in a single well-designed programming language (or perhaps a small number of stylistically varied well-designed languages) that imposes a minimal number of obstacles to the programming task. One should first learn to play baseball with a straight bat. Only later should one learn the specific workarounds needed for flawed languages.

I also believe that programming must be integrated with program specification. To document their programs precisely, programmers should have a thorough understanding of specification languages (which are just as much "programming languages" as are languages for execution). They should also be able to construct (most likely with computer assistence) formal proofs that programs meet their specifications.

It is becoming increasingly clear that this kind of integration is essential for the creation of highly reliable software. More to the point educationally, formal verification is the most effective way of teaching the nascent programmer to consider his work carefully enough to be sure of its reliability and lack of bugs.

An additional reason for teaching programming languages to programmers is that whenever they design a textual (or perhaps even graphical) input format that will affect the behavior of a computer, they are designing a programming language. It is often objected that such formats are not real programming languages since, for instance, they are not Turing complete. But the reality is that these formats are close enough to programming languages to be subject to the same design faults. And so, from designers innocent of the principles of programming language design, we have been given a nearly endless succession of "input formats" that seem to be designed to encourage error. (Flawed macro facilities that violate the basic laws of variable binding are particularly prominent examples.)

In summary, the teaching of programming and of programming languages are intrinsically linked. They should include specification and verification, and they should be taught with as much regard to rigor and precision as in traditional mathematics.

The last of these points deserves particular emphasis. The era of seat-of-the-pants programming and language design is waning, and the future belongs to programmers who are well-trained in the theory and logic that underlies their work.

This is not the place to try to lay out a specific curriculum. Instead, to indicate what should be expected of such a curriculum, here is a short and partial list of the kind of capabilities that should be expected of the competent computer professional:

- To write short error-free programs, without testing, for tasks such as binary search, and to use a formal system such as Hoare logic to verify them.
- To use language features such as higher-order functions and continuations to implement tasks such as backtracking and memoization.
- To write concurrent programs using shared variables or message-passing, and to give rigorous informal arguments for their lack of errors.
- To specify precisely and implement correctly a simple programming language.
- To design, specify, and implement correctly a simple language for a special-purpose input format.
- To hand-translate programs in functional or object-oriented languages into a low-level language such as C or assembly language.
- To write a clear and error-free program of sufficient complexity — say a syntax-directed editor — that it must be structured carefully.
- To estimate the time and space requirements of basic operations on widely used data representations.