??

Eduardo Bellani

October 1, 2009

1 Xurubemba

The problem that we will cover in this article is the one of learning how to think programmatically, that is, to program.

What we will be focusing is specially the demonstration of paths and technique's a single aspirant programmer would use to grasp not only the minor surface details of the craft, but its meaning and consequences, so that it could have a chance at approaching complex and dynamic problems. The same holds true for a teacher who wants to build a class for novice programmers.

There are a latent potential for an increase in the depth of the knowledge about computation in the general population. You can observe for instance it by the appreciation of computer games and graphics. Sure, not every gamer of graphic designer has a desire for becoming a programmer, not even the majority. But one can assume that from the enormous pool of people involved in those activities there could be an significant category interested in computing itself.

There are a lot of benefits that leaning to program brings to an individual, besides the possible monetary incentive. Sussman [2005] Not only that, but this mastery could lead to an increase in critical thinking and multiple levels of abstraction. As Felleisen et al. [2001] states:

> On one hand, program design teaches the same analytical skills as mathematics. But, unlike mathematics, working with programs is an active approach to learning. Interacting with software provides immediate feedback and thus leads to exploration, experimentation, and self-evaluation. Furthermore, designing programs produces useful and fun things, which vastly increases the sense of accomplishment when compared to drill exercises in mathematics. On the other hand, program design

> teaches the same analytical reading and writing skills as English. Even the smallest programming tasks are formulated as word problems. Without critical reading skills, a student cannot design programs that match the specification. Conversely, good program design methods force a student to articulate thoughts about programs in proper English.

But there is a lot on uncertainty as to the proper way to get educated. The reasons are plenty, but an educated guess would probably involve the following reasons:

1. Too much disconnected information, sometimes conflicting with each other
2. An emphasis on the final product and not the process in most of the "official" materials Harvey [1991]
3. A lack of explicit thinking tools for dealing with semantic complexity, and not syntactic detailFelleisen et al. [2002]

The contributions we try to accomplish in this article are:

1. A central and coherent roadmap one can use to become a competent programmer
2. A review of the literature involved in the roadmap, pointing out their perceived strengths and weakness
3. Some indications on where to go after the way is walked

References

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. *How to Design Programs. An Introduction to Computing and Programming*. The MIT Press, Cambridge, Massachusetts London, England, 2001. MIT – Massachusetts Institute of Technology – Online version 3^{rd} edition 22. September 2002: `http://htdp.org/2003-09-26/Book/curriculum.html` – last visited 1^{th} Oct 2009.

Matthias Felleisen, Robert B. Findler, Matthew Flatt, and Shriram Krishnamurthi. The structure and interpretation of the computer science curriculum, 2002.

Brian Harvey. Symbolic programming vs. the a.p. curriculum. *The Computing Teacher*, 18, 1991.

Gerald Jay Sussman. Why programming is a good medium for expressing poorly understood and sloppily formulated ideas. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 6–6, New York, NY, USA, 2005. ACM. ISBN 1-59593-193-7. URL http://doi.acm.org/10.1145/1094855.1094860.