JFP **13** (5): 835–838, September 2003. © 2003 Cambridge University Press
DOI: 10.1017/S0956796803004799 Printed in the United Kingdom

Welcome to the Educational Pearls Column

MATTHIAS FELLEISEN

College of Computer Science, Northeastern University,
Cullinane Hall, 360 Huntington Avenue, Boston, MA 02115, USA
(*e-mail:* matthias@ccs.neu.edu)

Abstract

Education matters. The lack of education on functional programming languages and techniques is visible on a daily basis. Our students, co-workers, friends, and colleagues just don't know enough about these ideas and therefore often fail to implement the best possible solutions for their programming problems.

The purpose of Educational Pearls is to address the education problem of our community from many different angles. It will include contributions on curricula issues, educational software support, and educational experiences. They will help teachers, professors, researchers, and software developers to promote functional programming languages and techniques in their respective contexts.

1 Education matters

Programmers ought to use functional programming techniques and even functional programming languages. Managers should understand how thinking in a functional manner improves their programmers' productivity for both the development of software as well as its maintenance. Alas, far too little software is built in a functional manner, and even less is built with functional programming languages.

Some programmers have never heard of functional programming. Some vaguely remember it from a course on comparative programming languages. If they do, they also recall that functional languages are about writing recursive versions of factorial and Fibonacci, and everyone knows that these functions can be implemented with loops anyway. None of them ever understood how one could possibly use a functional language for solving real-world problems. Worse, people "know" that functional programs are slow. After all, functional languages implement higher-order, lexically scoped functions, and such things make programming languages slow. Never mind that languages such as Perl and JavaScript support closures and are nevertheless widely used as systems and Web scripting languages. In short, people know little about functional programming techniques and languages and what they know is at best half-true.

From the perspective of our community, this diagnosis means that we have failed to educate students, colleagues, research partners, friends, and managers about the advantages of functional programming. "Education" means many different things. Education may happen in the classroom. There we may have to show that functional

languages are, for example, better scripting languages for Web programming than the currently popular ones. Education may happen in a faculty meeting. There we may have to explain how functional programming is useful for explaining concepts in a wide array of subject areas: hardware design, software design, network protocols, and so on. We certainly have to dispel the rumor that functional languages are for teaching compilers, interpreters, and type systems. Education may happen at the water cooler. There we may meet a colleague or a manager who has encountered a programming problem that we can solve easily with a functional technique. We may need to explain the idea in a conventional language but we must not forget to point out that it is built into the functional world.

2 Educational pearls

Education does not happen in the abstract. Good education requires concrete examples, experiences, and sometimes challenges to established beliefs. Because I believe that our community lacks an organized effort in this area, I am starting this series of Educational Pearls. The primary purpose of this new series is to publish examples, experiences, and challenges for all levels and for all kinds of education. Educational Pearls will address our education problem from many angles: the curriculum itself; individual courses; examples of how to hook students, colleagues, and managers; software tools; and success stories.

A *curriculum pearl* should explain and justify the development of an entire computing curriculum or a course (seminar, workshop). It should discuss the context (course, faculty colleagues, students), the content, teaching experiences, and conclusions for the community. For an example of historic interest consider the development and implementation of Abelson and Sussman's 6.001 course at MIT, now known by the title of their book *Structure and Interpretation of Computer Programs* (Abelson *et al.*, 1985). At the time, the course altered the landscape of introductory computing in a revolutionary manner. Abelson and Sussman showed that an introductory course could use a simple, but powerful functional programming language to teach deep concepts from computer science and not just the syntax of a first programming language. Similarly, the *Journal of Functional Programming* volume 3 issue 1 was dedicated to the problem of introducing functional programming into the curriculum.

When we are developing such new functional approaches to teaching computer science subjects, it will often help with the dissemination to other institutions if we describe the work and its background in a pearl. In this vein, Manuel Chakravarty and Gabriele Keller (University of New South Wales) will present their experiences with teaching Haskell at the introductory level and their challenging conclusion that "we should *not* teach purely functional programming in freshman courses." Starting from this very same thesis, I will describe the design rationale for my team's TeachScheme! curriculum effort and textbook (Felleisen *et al.*, 2001) in a parallel pearl.

An *example pearl* should illustrate the usefulness of functional programming with a concise, illuminating example. In other words, it shows the use of functional programming techniques as a pedagogical device or the use of devices that make

functional programming pedagogically attractive. Ideally, the pearl should include a description of the course context, assumptions about the student population, and a short experience section.

Harper's 1999 functional pearl on proof-directed debugging is an excellent example of this kind of pearl (Harper, 1999). His pearl explains how to develop a regular expression matcher and a correctness proof simultaneously. The failure to prove the required theorem suggests a revision of the specification, the program, and the proof. Harper developed this example for his second-semester course at CMU, and I consider it a beautiful illustration of how functional programming techniques help create solid software.

In a similar spirit, Chris Lüth will present a well-developed exercise for a first-semester course on Haskell in space. He shows how to develop a Haskell version of SpaceWar over a series of exercises. I will contribute a pearl on the design of functioning CGI scripts and Java servlets using continuation-passing style and other program transformation techniques from functional programming.

An *experience pearl* should tell success and failure "stories" about curricular as well as industrial efforts. For example, while researchers at both ATT and Ericsson worked on functional languages, one – SML/NJ – turned out to have no industrial impact at all and the other one – Erlang – conquered a company. We must learn from successful efforts what it takes to integrate functional programming into the real world, and we should also learn to avoid the pitfalls of failures.

A *software pearl* should present programming tools that help us teach functional programming techniques and languages. Even though our community may have the most elegant programming techniques and the best semantics for our languages, it certainly lacks adequate pragmatic support for program development compared to conventional languages and their visual integrated development environments. While many of us may not think of this lack as important, programmers and students do, and our community must address the problem.

Fortunately, there are already attempts to produce good program development environments or environment tools. Jung and Michaelson (Jung & Michaelson, 2000) proposed an interesting idea concerning the visualization of polymorphic type checking in this journal two years ago, and I am hoping to see educational pearls on teaching with types and type errors. My team's DrScheme programming environment is an example of a comprehensive pedagogic effort (Findler *et al.*, 2002); I intend to report on developments on this project on a regular basis. Serrano's Bee programming environment (Serrano, 2000) for integrating Scheme and C is important for our outreach to C programmers.

Reaching out to other language communities is indeed important. To have some impact on the real world and to stay relevant to computer science, our community must communicate to and with other language communities. We must explain how functional programming techniques help, even in an imperative or class-based (object-oriented) world. To provide an example of such an effort, the first educational pearl is a contribution by Hartel, Muller, and Glaser on functional experiences in C. In the next issue, they explain how they taught C as the first language using functional techniques.

3 Call for contributions

Educational Pearls is a new forum for and by readers. I am hoping that all of you will benefit from this new series in your daily work. I am also hoping that some of the articles that will appear here will help you communicate with people outside our community. Equally important, I also need your help. Many of you have experiences in educational contexts, be that with the curriculum, individual courses, modules for courses, professional workshops, or informal contacts. Many of you have experienced the successes and failure of functional approaches. I welcome your contributions. Please visit the home page for Educational Pearls at

`http://www.ccs.neu.edu/home/matthias/JFP/`

and contact me via email.

References

Abelson, H., Sussman, G. J. and Sussman, J. (1985) *Structure and Interpretation of Computer Programs*. MIT Press.

Felleisen, M., Findler, R. B., Flatt, M. and Krishnamurthi, S. (2001) *How to Design Programs*. MIT Press.

Findler, R. B., Clements, J., Flanagan, C., Flatt, M., Krishnamurthi, S., Steckler, P. and Pelleisen, M. (2002) DrScheme: a programming environment for Scheme. *J. Functional Prog.* **12**(2), 159–182. (A preliminary version appeared in *PLILP 1997: Lecture Notes in Computer Science 1292*, pp. 369–388.)

Harper, R. (1999) Functional pearl: Proof directed debugging. *J. Functional Prog.* **9**(4), 463–469.

Jung, Y. and Michaelson, G. (2000) A visualisation of polymorphic type checking. *J. Functional Prog.* **10**(1), 57–75.

Serrano, M. (2000) Bee: an integrated development environment for the Scheme programming language. *J. Functional Prog.* **10**(2), 1–43.