J. Functional Programming **12** (1): 89–90, January 2002.
DOI: 10.1017/S095679680100421X Printed in the United Kingdom

Book review

Principles of Programming Languages Design, Evaluation, and Implementation (3rd ed.) by Bruce J. MacLennan, Oxford University Press, 1999, ISBN 0-19-511306-3.

Principles of Programming Languages provides just over 500 pages of clearly written and incisive material, which is aimed approximately at the level of a second year undergraduate.

As a textbook, it would make a nice prelude to Aho, Sethi and Ullman's *Compilers Principles, Techniques and Tools* in a course which emphasises procedural programming languages, or to a more specialised book on implementing languages from the object, functional or logic programming paradigms. From my own perspective as a 'working programmer' with an interest in programming languages and software design, I enjoyed both the content and the style.

Professor MacLennan's book is based on 19 principles of programming languages which are given pole position on the inside front cover of the book and which maintain their prominence in the logical development of most chapters. A well-woven but narrow thread of computer language history illuminates his dissection of the eight languages which are discussed in depth.

Those languages are primitive 'pseudo-code' (one chapter), Fortran, (one), Algol-60 (two), Pascal (one), ADA (two), Common LISP (three), Smalltalk (one) and Prolog (one). There is also a chapter on the implementation of block structured languages, along with short introductory and concluding chapters. There is only shallow discussion of the popular languages in the C family (C, C++ and Java). Functional languages such as ML, Haskell or Clean are not mentioned. Visual and parallel (as opposed to concurrent) programming paradigms likewise fail to make an appearance.

On the other hand, you will find a three page description of the Interlisp development environment and a four page description of John Backus' FP functional language complete with examples and exercises. There is a considerable amount of functional programming material, but that is not what this book is about in general.

The exercises throughout the book are generally presented as essay topics or programming assignments and either fill gaps in the commentary (for example, covering the reports on Modula II and Oberon) or examine important language design decisions and trade-offs. They seem to be well thought out.

Practical advice on basic language implementation and evaluation abounds. There is even a brief note on how to help a language standardisation committee produce an elegant and manageable language design. Good stuff!

During the discussion of pseudo-code interpreters, the notion that programming languages have 'ampliative' and 'reductive' aspects is introduced. The model from which these terms come derives from the theory of the phenomenology of tool use and is one of the more interesting aspects of the book.

New tools tend to give us new and better ways of doing things (ampliative), but with the effect of reducing our capabilities in other ways (reductive). An ampliative effect of high level languages is the freedom given to programmers to focus on higher level aspects of the task at hand, rather than worrying about native data representations. A reductive effect of such languages is the additional distance from the hardware resulting in less low level control. (I knew a programmer who disassembled third party graphics libraries so that he could optimise function entry points for the processor at which we were aiming our product. Sometimes speed can be everything, even in very large programs.)

The ampliative/reductive duality often leads to people reacting with either fear or fascination to new technologies, and it can take time before a comfortable balance is reached. This is an important practical point for budding commercial programmers who must take account of how their end-users will react to new features and unusual user interfaces, not to mention hopeful language designers.

Professor MacLennan later develops themes derived from fields such as structural engineering design. His holistic approach contextualizes programming languages and is another great strength of this book – the principles flow from common sense and easily understood guidelines. Taste and elegance are common denominators which most of us can learn to appreciate eventually. They are certainly applicable to the design, evaluation and implementation of computer programming languages. Once appreciated, these principles can be carried over into other aspects of the software development process.

The structure and themes of this book also set it apart from others. Many books and courses confuse modern day programming language paradigms (such as object, functional or logic programming) with more general principles which computer programmers and language designers can use to build their own internal working model of computer languages (such as abstraction, structure, information hiding and simplicity).

I like this book, but I feel that there are several improvements to be made at little cost.

If my editor had forbidden me to add more pages, I would use the inside back cover to include an evolutionary diagram of 20 important languages, and the facing page for a table of the same twenty languages and 15 important language characteristics. In the Prolog chapter, the example code and discussion change several times between Clocksin and Mellish Prolog and the hypothetical pure logic language used to expose Prolog's darker corners. Those changes could be better delineated. The last few chapters refer less to the 19 principles, which may be the result of a trade off between orthogonality and simplicity, but could also signal a need for editing.

If I had four pages further, they would be partly used to show the driving forces controlling the evolution of the remarkable series of languages handed down to us by Niklaus Wirth (including Modula II and III and Oberon). I think that an in-depth discussion of Wirth's academic life and motivations would be of great interest and probably inspirational to students with an interest in computer languages. Many good tertiary level chemistry texts do this kind of thing, for example.

The remainder of those four new pages would briefly mention SML, Ocaml, Haskell, Mercury, Hindley–Milner type inferencing and the visual and parallel programming paradigms (for example, my old friend Thinking Machines Corporation's C*). Perhaps this could be done partly through exercises and partly through discussion.

Reading *Principles of Programming Languages* reminded me of days in the University of Queensland Engineering library browsing books on computer languages, 15 years ago. The language mix is very much set in the past. However, when I visited the same library to prepare this review, I was surprised to note that perhaps Professor MacLennan's decision to look in depth at ADA and Common LISP was a worthwhile choice after all – there was not one book on SML and only one on Haskell! There are pedagogical reasons for making these choices and Professor MacLennan clearly thought about what went into the book and what didn't. However, they may not sit very well with the superficial expectations of many second year computer science students or adherents of modern day popular programming languages such as C++. In the end, this is a minor criticism; you will need to make your own judgement.

I get the feeling that Professor MacLennan shows me something of his professional philosophy through this book and I think the world is a better place for it. I am pleased to have it on my bookshelf and I believe that his nineteen principles of programming languages should help you and your students to produce quality software which is, in the end, what programming should be about.

Mike Thomas