Essentials of Programming Languages (third edition) by Daniel P. Friedman and Mitchell Wand, MIT Press, ISBN 978-0-262-06279-4, 2008.

Since 1992 programming languages courses at numerous universities have been based on this classic textbook [5]. The book has now appeared in its substantially revised third edition, this time without one of the original authors, Christopher T. Haynes. The book gives a profound introduction to basic concepts of programming languages. Hardly any mathematical knowledge or skills are required, but the reader should be a reasonably proficient programmer with some knowledge of Scheme, the programming language used throughout the book. The concise explanations also require attentive reading. The book takes the following principled approach: Programming language are best studied by writing interpreters for small programming languages with particular features. Implementing interpreters provides deep understanding and motivation, because they can be run. Additionally the reader will also learn about writing well-structured programs. Thus the approach is rather different compared to other programming languages books [4, 7] that compare programming language features by studying programs written in several common languages. Although the book relates high-level concepts (e.g. function call) to low-level features (context, stack), the interpreters aim for clarity, not efficiency. Throughout the book the authors assume that programs that will be interpreted are given as abstract syntax trees; using a suitable Scheme package for scanning and parsing is explained only briefly in an appendix. Despite some overlaps this is not a compiler-construction book. The book provides a solid foundation for any further study of programming languages, from compiler construction to formal semantics and type theory.

The book starts with two chapters on how to systematically define data types and functions over these data types, using the same approach as *How to Design Programs* [1]. These techniques are used throughout the book, especially for representing the abstract syntax tree of a program and for the interpreter as a recursively defined function over this abstract syntax. The main body of the book discusses interpreters for various programming languages, starting with a very simple one in Chapter 3 and then adding one feature after another. Chapter 3 concentrates on expressions and how to handle variable bindings and their scope. Subsequently the facility for defining named procedures and finally recursive procedures are added. Chapter 4 gives several variants of mutable state and eventually discusses parameter-passing variations, as differences are mainly exposed by mutable state. Chapter 5 considers low-level control-flow issues by studying continuation-passing interpreters. It also shows how continuations enable easy implementation of exceptions and threads. Breaking the extend-the-interpreter pattern Chapter 6 describes a general continuation-passing style transformation for any program. Subsequently Chapter 7, 8 and 9 add a static type system, a module system and an object system with classes, respectively.

For the third edition all chapters were revised and reorganised, but there are three main changes: all Scheme definitions come with contracts, there is a new chapter on modules, and the continuation-passing-style transformation of Chapter 6 was replaced. Contracts describe input and output sets of procedures like static types; the book does not use the full expressibility of dynamically checked contracts of some Scheme variants [2]. The authors do not assume the presence of any contract checking at compile or runtime; contracts are just structured comments to clarify the meaning of procedures and variables. I was surprised that the relationship between contracts and types is not even discussed in the chapter on types. As module systems are an important part of programming languages and their complexity is often under-appreciated, the new chapter on module systems is welcome. Basic concepts are studied in three steps. The chapter starts with simple modules just providing procedures, followed by support for opaque type export, and finally adding module procedures, that is, functors. Whereas the preceding chapter defines type systems first through type rules which are then implemented by type checking/inference, the module systems have no such independent specifications, but are directly defined through large checking procedures. However, such compromises may be required to limit the chapter size. The new Chapter 6 lucidly presents a first-order compositional continuation-passing style transformation. Nonetheless it seems odd that this chapter describes how to transform any program into continuation-passing style, but it does not give a reason for doing so for any program other than an interpreter.

Despite new material the number of pages have hardly changed. The text is clear and concise. Scheme code makes up a substantial part of the book (it is available for download); whereas initially code is explained in detail, later chapters often contain large chunks of code with few comments. The book provides many exercises. The reader should note the statements on the back cover: "exercises are a vital part of the text", "the text explains [only] the key concepts", "the exercises explore alternative designs and other issues". Hence even if not doing them all, the reader should give each exercise at least some thought. On their own, without the support of a teacher, readers may find it hard get the points of all exercises. Within the book hardly any references are given. Instead, the reader needs to refer to Appendix A which gives a clear guide to the appropriate bibliography.

I miss one topic in this book: concurrency. To effectively use today's and future hardware, all programs will have to be concurrent. Section 5.5 discusses threads, but the focus is on easy implementation within a continuation-passing interpreter. Admittedly, giving concurrency a more prominent role within this book would have been hard, because there are so many distinctive concurrency models and it is yet unclear which ones will be used most. What should a concurrently implemented interpreter look like?

The foreword by Hal Abelson makes the promise that this book teaches the skills for "appreciating which language features are best suited for which type of application". However, the book does not compare language features for their suitability for certain applications. For example, the single paragraph in Chapter 9 comparing objects and modules only emphasises the similarities and differences in implementation.

While reading the book I have been wondering whether the chosen implementation language, Scheme, is the best choice for the purpose. The authors do not take advantage of S-expressions: scanner and parser are required to process the mixfix syntax of the implemented languages and a data type facility comparable to the algebraic data types of ML and Haskell is used throughout. When writing calculations of Scheme programs the authors also use non-standard parentheses (`>>` `<<`) and special syntax for environments to improve readability. Also for some interpreters a specification in the form of separate equations over the abstract syntax data type is given first, followed by the implementation using a `case` construct for decomposing the abstract syntax tree. Hence I conclude that a functional programming language with algebraic data types and pattern matching, such as a member of the ML family, would be well suited as implementation language for the interpreters of the book. Wadler's criticism of calculating with Scheme [6] still applies. The book shows calculations of interpreters, not eager evaluations. To improve readability of calculations, function arguments are often left unevaluated until they are needed. Such calculations are easy to read, but need to be written with care. However, using a non-strict language such as Haskell instead of Scheme would be problematic, as all interpreters except for the continuation-passing style ones then would implement non-strict languages as well. I was disappointed that the dependence of evaluation strategy of the implemented language on the evaluation strategy of the implementing language is not even mentioned [3]. Using a pure functional language such as Haskell would also pose problems as the book uses some side effects within the implementation language: the store for mutable variables is implemented via side effects, the continuation-passing style trampoline uses a global mutable variable and tracing via output statements is used by some interpreters to obtain a readable account of their work.

In conclusion, this is a sensibly updated edition that every undergraduate computer science students should study to genuinely understand established programming languages principles.

References

[1] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. *How to Design Programs: An Introduction to Programming and Computing.* The MIT Press, 2001.

[2] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 48–59. ACM Press, 2002.

[3] John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher Order Symbol. Comput.*, 11(4):363–397, 1998. Originally appeared in: Conference Record of the 25th National ACM Conference, 1972.

[4] Robert W. Sebesta. *Concepts of Programming Languages.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[5] Walid Taha. Essentials of Programming Languages (2nd ed) by Daniel P. Friedman, Mitchell Wand and Christopher T. Haynes, MIT Press, ISBN 0-262-06217-8, 2001. *J. Funct. Program.*, 13(4):829–831, 2003.

[6] P Wadler. A critique of Abelson and Sussman or why calculating is better than scheming. *SIGPLAN Not.*, 22(3):83–94, 1987.

[7] David A. Watt. *Programming Language Design Concepts.* John Wiley & Sons, 2004.