Book reviews

Trends in Functional Programming (volumes 1 & 2) by Greg Michaelson, Phil Trinder and Hans-Wolfgang Loidl (editors volume 1), and Stephen Gilmore (editor volume 2). Intellect Books, Bristol, 2001, 2002
DOI: 10.1017/S0956796803214878

Having been out of academia for a number of years, I jumped at the chance to review these volumes, seeing this as an opportunity to get back up to speed with what was happening with the functional programming community. To give you some background, the point where I left to join industry was just after the world started using monads to structure their functional creations and I wondered what had come along since then.

The two volumes actually draw together papers from the first two Scottish Functional Programming workshops. As such, the collected works read far more like conference proceedings, with each paper standing alone, than textbooks. However, the two volumes do succeed in bringing together papers under various headings:

- parallel systems and programming,
- type systems,
- architectures and implementations,
- applications,
- theory,

and provide substantial introductions to each chapter in the form of very readable prefaces. Overall, I was surprised to see the focus that the Eden language seemed to command through the volumes, and found that these papers did go some way to filling in the recent gap in my FP knowledge.

One novel paper was the very readable work by Mark Green and Ali E. Abdallah on interfacing Java with Haskell. This paper details the embedding of a Haskell (Hugs) engine inside Java using JNI; this is the kind of process that is needed if FP is to make headway in the real world. Whilst being entertaining and readable, the discussion did not include any discussion of how to interface to Java application servers (where such use of JNI is not appropriate), or of use in multi threading environments or J2EE technologies such as JCA.

Using FP as an effective parallel language provides the impetus for many of the papers in the two volumes. In volume 1, the paper by Klusik *et al.* discusses an optimization strategy for Eden that reduces the number of intermediate processes and Hernandez *et al.* introduce *PARADISE*, a profiler for Eden built on top of GranSim. Hayashi & Cole present a BSP-based cost-prediction system for an implicitly parallel skeletal-language that contrasts with Ballereau *et al.*'s high-level BSP programming language BSML.

In volume 2, parallelism is again explored by Loidl *et al.*, who compare performance between GpH and Eden; the Eden implementation is discussed in more depth by Klusik *et al.* The paper by Cope *et al.* introduces parallel algorithms to determine propositional satisfiability and Hidalgo-Herrero and Ortega-Mallén & Loulergue provide separate papers giving evaluation semantics for parallel functional programming languages, the latter revisiting topics from volume 1 by also describing a programming approach to BSP algorithms.

The well-trodden subject of types and type inference receives coverage in volume 1, where Gilmore explores the use of types to assess whether mobile code can be trusted. Type-errors

and their explanation are visited by Yang, and also by McAdam, who generalizes previously disjoint work on this topic to complement this. The paper by Widera & Beiele discusses an approach to typing that combines the benefits of strict and soft typing and defines the main component of a complete type-checker. In volume 2 strong typing under extended type-systems is covered by McAdam *et al.* and also by Widera & Beierle.

FP implementation is discussed by Walton, who provides an abstract machine for SML-memory management, and by Woo & Han in their discussion of the ZG-machine, a space efficient G-machine. Morazán & Troeger present the MT (pure Lisp subset) architecture, together with experiments with its storage allocation algorithm. In volume 2 the pros and cons of cloning are evaluated by Faxén, and optimizing transformations explored by Pareja *et al.*

Application of FP receives attention in the form of the generation of test scripts from specifications by Baker *et al.* and in a functional design framework for Genetic algorithms by Rabhi *et al.* This topic reappears in volume 2, where Brown *et al.* compare Haskell and SML as languages for the implementation of GA. Sérot uses CAML to define higher-order polymorphic graph patterns which are then compiled to data-flow graphs, and Curtis applies FP to the combinatorial problems found in patchwork quilting.

Theory and reasoning attract the coverage of Dosch & Wiedemann, who propose a new class of list homomorphisms with accumulation and indexing as skeletons for data-parallel programming, and Lämmel discusses increased reuse from various program manipulations that are normally carried out manually. Baker-Finch describes an abstract machine for parallel lazy evaluation based on Sestoft's sequential abstract machine.

In conclusion, if, like me, you want a view of recent thinking and developments in functional languages and systems then these volumes have something to offer.

Chris Angus

> *Implicit parallel programming in pH* by R. S. Nikhil and Arvind, Morgan Kaufmann, 2001. DOI: 10.1017/S0956796803224874

This 500-page textbook is a complete, self-contained introduction to non-strict non-sequential functional programming in pH, a variant of Haskell. It is based on the authors' long experience of research and teaching in this area, and is aimed at advanced undergraduates, graduate students and programmers in general. The book assumes a familiarity with sequential imperative programming but this is not a formal prerequisite. A good basic culture in Computer Science is however needed to follow the many detailed explanations and examples dealing with semantics and efficiency.

The pH language is structured around a functional core with implicit multithreaded semantics. The core is extended with two increasingly complex and expressive forms of shared data: the deterministic single-write-multiple-reads I-structures and the nondeterministic M-structures.

The book follows this hierarchy and begins with a 7-chapter introduction to pure functional programming in pH, followed by three chapters on the shared data-structures. It ends with two short appendices (an introduction to the lambda-calculus and an operational semantics for pH) followed by a bibliography and a long index. Many programming examples are given throughout, their use of language features and exact semantics are commented on. Exercises are also presented, many based on the running examples.

The book is an excellent basis for an introduction to non-strict functional programming, but complete beginners could be confused by the frequent sections and remarks on precise semantics and expected efficiency of programs. In my experience with this kind of course, students have preferred a complete exposition of the pure-functional language before studying its semantics, along the lines of Cousineau and Mauny's ML-based book, *The Functional Approach to Programming*, 1995. However, an overall comparison between the two books is not possible: Nikhil and Arvind's book is broader in scope and its functional subset being

based on Haskell, it contains introductions to type classes and list comprehensions which Cousineau and Mauny's does not.

The imperative or ML programmer will notice with interest that pH loops do not impose complete sequencing of their steps and are thus maximally parallel in analogy to the PARDO of certain explicitly-parallel languages. A pleasant detail of pH syntax is the notation for loop index ranges "i ← [x0, x1 .. xn]" where the increment x1-x0 is given implicitly. The attribution of syntactic meaning to program layout was an unpleasant surprise to me as I would expect this to be a source of programming errors for many students. But that is a minor point.

Arrays are introduced in chapter 7 as direct access structures, necessary only for their efficiency and implemented with I-structures. A wavefront calculation is given as an example and the absence of redundant computations (with combinatorial explosion in a naive implementation) is related to nonstrictness and the I-structure semantics to be explained in chapter 9. I find this particular example very instructive and an excellent motivation for the I-structures. It has also been given as canonical example for the Caml-Flight (deterministic-parallel extension of Caml-light) of Foisy and Chailloux where pipelining was automatic, leading to a systolic execution of the wavefront calculation. I suspect that systolic execution in pH requires non-deterministic M-structure programming and would have been curious to see this issue discussed in the book.

Matrix multiplication is used as parallel program example where a trivial version is converted into a blocked program to improve "granularity" i.e. the computation vs communication (or memory access) ratio. This section ends with comments to the effect that compilers are able to realise this program transformation and that "the extra complexity of blocking is not something the programmer need worry about". Perhaps this is true of many algorithms but it will not convince the expert parallel programmer to trust the pH compiler in the presence of data-dependent algorithms, difficult load balancing or heterogeneous execution environments. This leads to the more general question of how much and how should the programmer interact with the parallel execution's key parameters like granularity, memory/communication bottlenecks etc. Expert parallel programmers who are usually very algorithm-aware will probably dislike the pH philosophy of completely hiding such factors.

Chapter 8 discusses and motivates sequencing, monadic I/O and parallel I/O as basis for the use of shared updatable data cells. I found its content difficult to master which is natural because it deals with synchronisation, and is only intended as lower-level layer for the I-structures and M-structures described in chapters 9 and 10.

I-structures are in my opinion pH's most original feature as they constitute a compromise between the simplicity of pure-functional programming and the expressive power of non-deterministic concurrent programming. Their semantics is clean and simple: an I-data (which could be an algebraic type's field, or an "I-array"'s element) is declared and allocated in one program part and its value is set by another, unique, program operation. The threads corresponding to other parts of the program where this unique value is needed are blocked until the value is set. The type system makes I-structures explicit and enforces the above deterministic protocol for their use.

The wavefront computation is a good example of the I-structure's interest. If the array being computed was programmed as a recursive function, say f: Nat * Nat → Float, with multithreaded semantics then f(n,n) would issue calls to f(n-1,n) and f(n,n-1) which would independently call f(n-1,n-1) twice, and the process would repeat itself thus leading to a combinatorial explosion of threads. But programming this as an I-array will lock the two "consumer" threads for f(n-1,n-1), namely the "producer" threads of f(n-1,n) and f(n,n-1), until the value is produced thus preventing the redundant subcomputation. As shown by this example, the I-structure paradigm can combine the simplicity of a top-down recursive definition with the efficiency of a bottom-up iterative algorithm. Chapter 9 of Nikhil and Arvind's book gives more examples and arguments in favour of I-structures, as well as formal semantics.

Chapter 10 describes general concurrent programming with M-structures based on a shared-memory put-and-take abstraction. I will not comment on it here as it does not constitute an original paradigm and introduces all the usual difficulties, and the expressive power of concurrent programming.

My main regret with the book, and with the pH project, is that it does not take into account two streams of research that have evolved during the 1990s and are relevant to the efficient implementation objective which the authors state in their conclusion. The first one is the algorithmic skeletons paradigm introduced by Murray Cole where specific higher-order functions are given well-understood parallel semantics with known complexities and used to express parallel algorithms by declarative programs. The other is the cost-model approach to parallel algorithms where execution models like BSP and LogP provide simplified architecture parameters to be combined with the parallel semantics of language operations to yield portable formulas for estimating execution time. Clearly, the pH philosophy is to consider cost models as a low-level implementation issue. But the skeptical parallel programmer would answer that without a cost-model it may be impossible for him to select alternate equivalent programs and thus improve the compiler's chances of producing scalable code. In the context of non-strict semantics these issues are even more difficult to tackle and I found it disappointing that, despite many comments relative to program efficiency, not a single mention is made of performance measures and performance parameters for current or future pH implementations.

Despite this criticism of implicit data placement, it is only fair to agree with the authors that most application programmers will never want to write explicit parallel algorithms. Research should therefore provide them with tools for easy generation of massive parallelism and automate as much as possible the remaining aspects of parallel execution (load balancing, bandwidth management etc). In this context the existence of a language like pH is encouraging as it will allow functional programmers to experiment with parallel and concurrent programming in the well-known and modern framework provided by Haskell. I recommend Nikhil and Arvind's book to all users, students and teachers of non-strict functional programming but I urge interested researchers to investigate the inclusion of parallel cost-models and algorithmic skeleton libraries to pH systems. Parallel programmers and users/researchers of strict functional programming languages could find it a useful and precise reference for documenting original research at the intersection of declarative programming and high-performance computing.

GAÉTAN HAINS
LIFO, Orléans, March 2002.

Structure and Interpretation of Classical Mechanics by Gerald Jay Sussman and Jack Wisdom with Meinhard E. Mayer, The MIT Press; 2001, ISBN 0262194554. DOI: 10.1017/S0956796803234870

'*Structure and Interpretation of Classical Mechanics*' is an elaborate, high-level textbook aimed at physicists and engineers: mainly teachers and their students. The authors concentrate on Lagrangian and Hamiltonian frameworks for the construction and analysis of equations of motion. They discuss also rigid-body dynamics, the theory of canonical transformations, as well as giving an introduction to the theory of perturbations. They show many interesting examples of highly non-linear dynamics, not often included in standard textbooks, but which recently became notorious in the context of 'Chaos' theory. The book, based on a MIT course delivered by the authors, is written in a personal, vivid style, with plenty of culture and occasional dashes of humour, which help to make its dense mathematics more digestible. The number of footnotes, often digressive, in the first chapter may be considered exorbitant by some readers, but others may like it. Currently, the book is accessible on-line through the personal pages of G.J. Sussman, and on the MIT Press site.

The authors underline the computational approach to their subject. They insist on writing computer-implementable formulae, and they produce many examples of programs written in the MIT dialect of Scheme, augmented by their own contribution, the software package 'scmtools' containing many numerical, symbolic and graphical procedures. Their programs are written in a typical Scheme functional style, and many formal derivations in the text follow the same philosophy. Thus, the book may be useful also for computer scientists, as a specific, practical application of functional programming methodology within the realm of science. We see there a dynamic world represented as a lazy stream of states, and plenty of higher-order functional combinations. The package can be used as a pedagogical tool, showing how to implement overloaded arithmetic, and how to deal with vector algebra in a functional style.

The reference of the title to the 'Structure and Interpretation of Computer Programs' is well illustrated by the style of the text, but a potential reader should not be confused: this is a book about physics; the computational side of it is secondary, although well developed and exposed!

As was mentioned earlier, the mathematical side of the book is rather involved, although it avoids such modern tools such as differential forms and categorical reasoning. This seems to be a conscious, pedagogic decision of the authors, who visibly preferred to avoid 'abstract mathematics', and to adapt the mathematical level of the presentation to their students. Their programs deal with numbers and explicit vectors, specifically data structures with 'up' and 'down' constructors which represent contravariant and covariant vectors; the authors make no attempt to define the dual basis as functionals over the other one. They also introduce functional objects, which may of course be applied, but also treated as symbolic entities. The package is equipped with generic arithmetic, and is able to compute derivatives of functions, and in general to deal with 'unevaluated' expressions: applications of functions to indeterminates.

This symbolic approach, easy to implement in Scheme, is powerful, but contains some traps. When the authors say that `(define (helix t) (up (cos t) (sin t) t)` may be 'just' written as `(define helix (up cos sin id))`, a casual reader may instantly lose the ability to answer such questions as 'what is `helix`?', or 'what kind of object produces `up`?'. Paradoxically, as early as the first chapter, the authors warn the reader against unclear and ambiguous mathematical notation! In this, and some other contexts, a reader oriented towards functional programming in general might have some warm thoughts directed at the strong typing discipline.

Derivatives seem to be calculated symbolically (syntactically), and one may wonder why the authors did not try to apply automatic differentiation technology to their derivations, since in most cases they need to differentiate functions in order to use them in further computations, rather than to show the derivatives as the final product. In general they are interested in computing the actual trajectories, and other distributions in the phase space, rather than just showing the structure of some formal entities, and that is the reason why their computational approach is particularly fruitful.

The MIT extensions to Scheme are heavily exploited, and two-level function definitions such as `(define ((transform frame) state) <body>)` are abundant. It is nice and readable to separate the 'static' and 'dynamic' function parameters, but an obvious reaction of somebody acquainted with other functional languages is: 'oh, they really need a language with currying!'.

The authors' programs will not run (directly) under other popular dialects of Scheme, which might weaken the popularity of the book a little. The MIT Scheme package 'scmtools' is also freely available (together with the MIT Scheme system; it is only installable under Unix).

So, here is our conclusion: this is a nice and useful pedagogic experience, an unusual approach to teaching classical mechanics with the aid of programming in the functional style. In order to obtain full satisfaction, the reader must possess some basic knowledge in both the domains concerned: physics and computing, or must really want to learn them both. From the perspective of functional programming the book suffers from some small flaws

of readability, resulting from the fact that the mathematical discipline of the authors gets weakened by the power, universality, and fuzziness of their Scheme programs. We love and we use Scheme, but we would be delighted some day to see such a book based on Haskell (or on ML), since in our opinion a polymorphic but strong type system is better suited to represent physical entities. Moreover, currying and partial applications are comfortable tools to generate abstractions (closures), since for beginners they are easier to use than the lambda constructs.

JERZY KARCZMARCZUK

Algorithms: A Functional Programming Approach by Fethi Rabhi and Guy Lapalme, Addison-Wesley, 1999, ISBN 0-201-59604-0, xi + 235pp.
DOI: 10.1017/S0956796803244877

It is perceived wisdom that the future success of functional programming requires a greater presence and influence within computer science curricula. Such a presence requires the usual support materials such as efficient compilers and appropriate textbooks. There are now many 'how to' textbooks (as there are in every paradigm) which typically describe how to program using a particular functional language and form the basis of an introductory course on functional programming. Less in evidence are the 'Advanced Topic in Computer Science: The Functional Programming Way' textbooks. *Algorithms: A Functional Programming Approach* is therefore a welcome addition to the set which includes *Modern Compiler Implementation in ML* (Appel, 1998), *Discrete Mathematics Using a Computer* (Hall & O'Donnell, 2000) (not an 'advanced topic', but the influence is there!), and *Purely Functional Data Structures* (Okasaki, 1998).

This book is structured like any classic algorithms textbook, and thus provides adequate support for an algorithms course, assuming that functional programming has been studied as a prerequisite. This is a refreshing addition to the plethora of algorithms textbooks, which either use an (imperative) pseudo-language to illustrate the algorithms, or implement them in one of several object-oriented or imperative languages.

The first part of the book delivers some scene setting and some building blocks to be used later in the text. It begins with a brief introduction to algorithms and how to measure their complexity. This is followed by an overview of functional programming in Haskell, which is the language used to implement the algorithms. The efficiency of functional programs is discussed in Chapter 3, with a review of the alternative reduction strategies. In their review of time efficiency analysis, the authors somewhat honestly admit that such analysis "under lazy evaluation requires more sophisticated techniques which are beyond the scope of this book". Therefore, all future analysis is based on strict evaluation, and thus the reader would need to seek other sources to deal with lazy evaluation. This is in contrast with Okasaki's text, where a framework for analysing lazy data structures is presented.

Chapter 4 introduces concrete data types and describes the application of well-documented approaches to transforming (for efficiency) functional programs defined over these types. The focus here is largely on efficient algorithms over given types, rather than the development of efficient data structures, as in Okasaki (1998). This is naturally followed by a discussion on Abstract Data Types (ADTs), and how they benefit algorithm development. The collection of examples is appropriate for an algorithms text, as is shown by their frequent use in the development of programs in later chapters. For example, in Chapter 7 the ubiquitous sorting algorithms are presented and heapsort is defined using a priority queue ADT, which for efficiency is implemented using a heap. Each algorithm in this chapter is accompanied by some space and time complexity bounds, but without any derivations.

Although graphs and their related searching, sorting and spanning algorithms are not new additions to functional programming texts (even though they are not included in Okasaki (1998)), Chapter 7 provides material beyond that covered in most introductory texts. Various

ADT representations are presented, although each is simply an implementation of a classically imperative graph representation. A second edition of this text could make use of Erwig's functional graph library (Erwig, 2001).

The penultimate chapters emphasise the inherent productivity of higher-order functions. Divide-and-conquer, backtracking search, priority-first search, greedy algorithms and dynamic programming are each defined using a dedicated higher-order function, with the latter used to implement algorithms to solve classic problems such as chained matrix multiplications and the shortest path. In all of these implementations, significant use is made of the ADTs defined earlier in the text.

Finally some *advanced* topics such as concurrency, monads and parallel algorithms are discussed in the light of algorithm development.

In conclusion, this book does not pretend to be an authoritative description of efficient functional data structures, nor does it provide a thorough overview of complexity analysis. However, it is an important addition to a small but growing collection of (functional programming) books which could be used on courses other than introductory programming courses. That is, it could certainly be adopted as the essential textbook on an algorithms and data structures course. A second edition would benefit from more careful proof reading to remove the (often case-based) errors in the Haskell code. Each chapter is terminated by a selection of appropriate exercises and some pointers to related publications. The index is thorough and a couple of brief appendices provide details on sources of Haskell implementations and some required mathematical baggage.

References

Appel, A. (1998) *Modern Compiler Implementation in ML*, Cambridge University Press.

Erwig, M. (1996) Inductive graphs and functional graph algorithms. *J. Functional Programming*, **11**(5), 467–492.

Hall, C. and O'Donnell, J. (2000) *Discrete Mathematics Using a Computer*. Springer-Verlag.

Okasaki, C. (1998) *Purely Functional Data Structures*. Cambridge University Press.

D. RUSSELL

> *Essentials of Programming Languages (2nd ed)* by Daniel P. Friedman, Mitchell Wand and Christopher T. Haynes, MIT Press, ISBN 0-262-06217-8, 2001. DOI: 10.1017/S0956796803254873

This textbook is an introduction to basic programming language concepts, and covers topics including datatypes, functions, objects, typing and type systems, and implementation techniques. The book explains many key algorithmic aspects of programming language design such as type checking and type inference, but without dwelling too much on less-obviously algorithmic properties, such as principality, for example. As such the textbook is accessible to the beginning undergraduate student, and assumes little or no experience with mathematics and formal logic other than familiarity with basic set theory. At the same time, the book's careful and detailed treatment of various language concepts provides what I view as crucial motivation that a student needs in order to pursue more abstract treatments of programming languages, such as in courses on semantics and type theory.

Whereas Reade (1989) introduces programming in a functional language (ML), and Gunter (1992) and Nielson and Nielson (1992) introduce mathematical semantics, this book assumes knowledge of a functional language (Scheme) and introduces programming languages concepts that are easily identified by the expert as mathematical, but without exposing the reader to mathematical formalisms. Compared to MacLennan's text (MacLennan, 1987), this book has more hands-on treatment of how languages can be implemented, more of a focus on

notions easily identified with semantics concepts, and less attention to the historic evolution of programming languages.

The choice of Scheme brings up the question of what is 'the ideal' language for such an introductory course. On one hand, a key advantage of Scheme is that it promotes functional programming, a style which is now widely viewed as useful in implementing languages. It is also refreshing to see, once again, how easy it is to experiment with language implementation in Scheme, and to take advantage of the convenience of s-expressions as a simple and uniform tree-based representation for data. On the other hand, the absence of static typing could be viewed as a weakness. This does not seem like a real problem, as there are statically typed variants of Scheme (Findler *et al.* 1997).

Chapter by chapter review

The book is divided into eight chapters. Each chapter ends with a set of bibliographic references to key sources that the interested student can look up to probe further into the subject covered by the chapter.

Chapter 1 aims to show a student who has a basic background in programming and mathematics that there are strong connections between the two. For example, both BNF and structural induction are introduced in this chapter, and the connection between the two is made explicit. As an undergraduate student I was introduced to BNF long before I became familiar with structural induction. As with all the following chapters, this one has plenty of interesting examples and even more exercises. The number of examples in each chapter is may be small, but each example is explained with care and in great detail.

Chapter 2 introduces data abstraction and abstract datatypes. The chapter introduces one of the book's many innovative pedagogic tools, in this case a utility (macro system) for introducing ML-like datatypes. The chapter also discusses the notion of interfaces within programs, and stops short of entering into the discussion of various formal tools for specifying interfaces such as types, specification languages, and module.

Chapter 3 introduces the concepts of expressible and denoted values, and the first in a sequence of languages studied in the book is introduced. This 'starter kit' language has arithmetic expressions, conditionals, and let statement. Once the basics of implementing this language are explained, the language is extended with the notions of statements (including sequences and blocks), and then procedures and closures. The implementation of call by need and call by reference are treated, and a thorough exposition of the implementation design space is presented.

Chapter 4 introduces types and type systems. The focus is on simple types where base types and (partial) functions are the key constructors. The idea of algorithmic (or automatic) type checking is introduced, and an example implementation is presented and discussed in detail. The mathematical notion for defining rules for typing derivations is carefully introduced, thus preparing the student for reading more formal treatments of typing. Somewhat surprisingly, references in this chapter do not include pointers to the work of Wright and Felliesen (1994), the tutorial papers by Cardelli (1991, 1997), or Hindley's book (1997). Fortunately, this is easily remedied by the instructor and in future editions of the text.

Chapter 5 gives four implementations of a language with objects, classes, inheritance, and polymorphism. Each implementation is a (more efficient) refinement of the one before it. The principles behind each refinement are quite pervasive, and as such make for valuable examples of the idea of developing more efficient programs by refinement. The next chapter is a natural extension to this one, and discusses type checking this language and gives detailed examples of how these types can be used in optimization.

Chapter 7 explains the concepts of a control context and tail recursion, and contains a progression of more efficient interpreters. I have used intuitions presented in this chapter in explaining continuations to students in an advanced graduate course that I am currently teaching. Interpreters are presented for languages with exceptions and multi-threading, and for

a simple logic programming language. A detailed explanation of the technique of trampolining is given. The idea of refining interpreters is here taken to the point where the interpreter is 'registerized' so as to use only assignments, simple conditionals, and gotos (tail calls with zero arguments).

Chapter 8 introduces the more advanced topic of the Continuation-Passing Style (CPS) translation, and discusses various approaches to implementing it.

References

Cardelli, L. (1991) Typeful programming. In: Neuhold, E. J. and Paul, M., editors, *Formal Description of Programming Concepts*, IFIP State-of-the-Art Reports, pp. 431–507. Springer-Verlag.

Cardelli, L. (1997) Type systems. In: Tucker, Jr., A. B., editor, *The Computer Science and Engineering Handbook*. CRC Press.

Findler, R. B., Flanagan, C., Flatt, M., Krishnamurthi, S. and Felleisen, M. (1997) DrScheme: A pedagogic programming environment for scheme. *Programming Languages: Implementations, Logics, and Programs*, **1292**, 369–388.

Gunter, C. A. (1992) *Semantics of Programming Languages*. MIT Press.

Hindley, J. R. (1997) *Basic Simple Type Theory. Cambridge Tracts in Theoretical Computer Science 42*. Cambridge University Press.

MacLennan, B. J. (1987) *Principles of Programming Languages: Design, Evaluation, and Implementation, 2nd ed*. Holt, Rinehart and Winston.

Nielson, H. R. and Nielson, F. (1992) *Semantics with Applications: A Formal Introduction*. John Wiley & Sons.

Reade, C. (1989) *Elements of Functional Programming*. International Computer Science Series. Addison-Wesley.

Wright, A. K. and Felleisen, M. (1994) A syntactic approach to type soundness. *Information and Computation*, **115**(1), 38–94.

WALID TAHA

The World of Scripting Languages by David W. Barron, John Wiley & Sons, 2000, ISBN 0-471-99886-9. DOI: 10.1017/S095679680326487X

What is a scripting language? There are many answers to this highly subjective question. David Barron offers his definition, along with a guide to half a dozen or so mainstream scripting languages in this 492 page survey. The writing style is informal, but plain.

The book begins with a short discourse on the nature of scripting languages, what they are used for, and how their role has changed over the years. The author touches briefly on Unix system administration, COM Automation, CGI, and HTML Document Object Model scripting.

Professor Barron chooses a decidedly strict definition of a scripting language, emphasizing scripting languages as 'glue' for connecting or automating components written in more traditional languages. There are problems with this strict definition, as he admits, such as the fact that many large systems are built entirely out of so-called scripting languages, with no 'traditional' language components at all.

The larger problem with this definition is that it leads the author to discuss a somewhat odd selection of languages. For instance, Python and Dylan are dismissed from consideration because they are 'alternative programming languages', not scripting languages at all. However, Visual Basic is included in the book, on the basis that it scripts COM components. It is not clear to me how Python is different enough from Perl and Tcl to be put in an entirely different category, and the author gives no explanation for the distinction. In the same vein, one questions whether a compiled language like Visual Basic can really be called a scripting

language. Finally, in a book whose title promises the whole world of scripting languages, It is a little disappointing to see only the most mainstream of languages covered. What about Lua (designed from the ground up as a scripting language), Scheme (scripting/extension language of choice for the GNU project), and many others?

The first language discussed is Perl, which is given a credible introduction in 99 pages. The text is appropriate for a programmer who is already familiar with another language, and the section is filled with concise explanations of Perl features, followed by a number of short examples of common tasks. This is easily the strongest section in the book.

The next 91 pages go to Tcl and Tk, along with shorter sections on Perl-Tk, and Java-based Tcl relatives Jacl and TclBlend. The section includes a number of Tk examples. It would be interesting to compare Perl's idiosyncratic syntax and semantics with Tcl's almost total lack of both, but the author abstains from comparision or discussion, citing a wish to avoid religious issues.

Visual Basic is the subject of the next 82 pages. This section shines, because in addition to giving a reasonable summary of the language, it drags out quite a number of VB's more baroque and little-used features for display. Any reader who is even slightly familiar with VB will get a good chuckle out of this.

After these chapters, the focus shifts to the web, beginning with shorter (30 pages), but still reasonably thorough coverage of Javascript (with notes and sidebars about JScript and ECMAScript). After discussing Javascript, the book turns to VBScript, and here the author makes a startling decision: rather than quickly treat the differences between Visual Basic and its stunted cousin VBScript, a substantial amount of the earlier material is simply repeated, with marginal rules to indicate the places where the text has changed. The intent is for the chapter to stand on its own, without reference to other chapters, but to me it seemed excessively redundant. Since VBScript is so very similar to VB, a few short paragraphs added on to the end of the Visual Basic section would have given generous coverage.

The rest of the book is dedicated not to scripting languages themselves, but to the object models which they script. The breadth of this material is considerable, amounting to a whirlwind tour of DHTML and the Document Object Model, the Microsoft Office object model and COM Automation, and wrapping up with the Microsoft Scripting Runtime Library and the Windows Scripting Host. This is all good introductory material. The book ends with some interesting historical background, including brief sections on such worthies as REXX, AWK, and even OS/360 JCL.

On the whole, the material is sensible and covers a large range of technologies in limited space. As a survey of mainstream scripting languages and their object models, the book is effective, though it does not include enough material to serve as a tutorial in any one of its subjects. It is unlikely that experienced practitioners will find much of interest here, but it would make a decent introduction to the concept of scripting languages for students or those who have been involved in altogether different pursuits.

Bryn Keller