of concurrency, synchronisation and communication. Other aspects considered include its application to embedded and distributed systems, and implementation issues.

The book is structured as follows. The introductory chapter classifies the various approaches that concurrent programming languages have adopted to handle process execution, synchronisation and communication. Chapter 2 gives a very brief overview of the Ada language, followed by a description of the Ada tasking model. Chapter 3 summarises work undertaken on formal aspects of the model, such as formal semantics and proof systems. Chapter 4 is a detailed assessment of Ada as a general purpose concurrent programming language, using the classification of Chapter 1. Chapters 5 and 6 consider particular issues relating to embedded and distributed systems. Implementation of Ada tasking is discussed in Chapter 7, and proposed changes to the model are evaluated in Chapter 8.

With 234 references, the authors succeed in their aim of comprehensively reviewing the Ada tasking model. There are no major omissions or errors: the book was completed before the 1987 Ada-UK/SIGAda International Workshop on Real-Time Ada Issues, the proceedings of which (Ada Letters VII (6), Fall 1987) further elaborate many of the topics concerning embedded and distributed systems.

The main benefit of the book is probably in highlighting recognised problem areas with the tasking model. As a balance, perhaps more could have been said about the rationale for some of the tasking constructs, and also about the advantages, such as portability and stability, gained from having an ISO standard language definition held unchanged for a decade.

Some familiarity with concurrent programming in general is assumed on the part of the reader, and a previous knowledge of Ada is required to fully understand the algorithms given in the appendices. Hence, for a first introduction to Ada tasking, a publication such as Alan Burns' "Concurrent Programming in Ada" (CUP 1985) is more appropriate than this one, which will be of most interest to users and implementors of embedded or distributed Ada systems, and to programming language researchers. To this comparatively restricted audience, this book can certainly be recommended.

Ian MEARNS
Marconi Data Systems
Chelmsford
United Kingdom

Computability Theory, Semantics and Logic Programming. By Melvin Fitting. Oxford Logic Guide 13, Oxford University Press, Oxford, 1987, Price £25.00 (hard cover), ISBN 0 19 503691 3.

Many books invent a new language for pedagogical purposes. This book invents a whole panoply of them both declarative and imperative. The purpose is not, as

usual, solely to cudgel the reader into adopting a particular paradigm but to explore the foundations of computation by implementing one in another. The book exemplifies that we have no choice over what can be computed; the only choice we have is the way we choose to compute it. The bias of the book is that Logic Programming is *the* way.

With this stance it polarizes its potential audience about an immutable demarcation line. It is strange how a discipline with logic in its name attracts such irrational feelings (both fanatically hostile and zealous, particularly among university professorial staff). However, coming across this book on library or bookstore shelf most Prolog devotees would not immediately recognise this as a book on Logic Programming, apart of course from the title. To begin with, a Prolog clause in Edinburgh syntax (with arguments suppressed)

$$p :- q, r, s, t.$$

would be written in this book

$$T \rightarrow S \rightarrow R \rightarrow Q \rightarrow P$$

where the logical implication $\rightarrow$ is assumed to be right associative. While the replacement of $:-$ by an implication symbol is commendable, Fitting's syntax for Horn clauses is reminiscent of programs written on teletype terminals where everything has to be in upper case. I malign the syntax somewhat as only predicate symbols are in upper case but as you have by now appreciated I find it an annoyance that distracted from my appreciation of the book.

One of the endearing features is that at the end of each Chapter there is a background section providing references to the original literature. It is interesting to note that Fitting appears to attribute Horn Clauses to Smullyan. (Horn clauses have been attributed to people other than Horn and indeed I understand that Horn is not a little embarrassed by having them named after him.)

Another difference between Prolog and this book is that Prolog is concerned with backwards reasoning (modus tollens) while a family of related languages, EFS (elementary formal systems), introduced in the first two chapters is concerned with forward reasoning (modus ponens); in theorem proving circles this has been called hyperresolution. The different members of the family EFS are distinguished by their data structures. In the second and third chapters Fitting seeks to establish semantics for the members of EFS in terms of minimal models and least fixed points. In this respect the book can, naturally, be compared with Lloyd [1], the as yet only other book which considers the semantics of Logic Programming.

Fitting describes the fixed point theory of the EFS family in terms of operators: functions which map relations to relations while Lloyd views the semantics of Logic Programming from the more abstract setting of lattice theory. From personal teaching experience Fitting's approach is much more accessible to students. Fitting defines a semifixed point of an operator T as a subset S of its domain such that $T(S) \leq S$ (set inclusion). Lloyd doesn't define this concept but uses it in passing. I believe it

is worth naming such sets as they turn out to be models of the program. Fitting also defines a notion of compactness which is used in establishing continuity of the immediate consequence operator. Again Lloyd proves that the immediate consequence operator has this property without naming it. It is such devices for breaking up, what appear to students complex proofs, which make Fitting's book suitable for self study; Lloyd makes no concessions to the mathematically immature. Fitting's proofs are simple and straight forward which make the book a suitable text for third year and possibly second year undergraduates while Lloyd, I believe, is strictly a graduate text. Furthermore, there are no vital missing steps in Fitting's proofs; the same cannot be said for Lloyd's book. My feelings about Lloyd's book has been expressed by Lassez et al. [2] in relation to unification... those who do not realise the subtleties are often satisfied with a casual understanding, this leads to incorrect statements and when some flaw is expected in establishing an "intuitively obvious" result, the proof is left to the reader.

From the pedagogical point of view Fitting's book is to be preferred to Lloyd's. However, Fitting doesn't really consider negation (as beyond the scope of the book) while Lloyd's book is mainly concerned with the semantics of negation as finite failure (this is where the mathematically interesting results are to be found). So to a large extent the comparison between the two books is unfair; Fitting's book is pedagogical and mainly concerned with establishing concepts of computability, while Lloyd's is a compendium of results on Logic programming. Lloyd [1] only devotes 3 pages and a single theorem to computability. Whereas Lloyd shows that Horn clause programs are capable of computing partially recursive functions, Fitting carefully builds up the argument that recursively enumerable relations are sound basis on which a theory of computation can be defined.

To this end Chapter 4, on implementing data structures, takes a categorical viewpoint with data structures as objects and implementations of one in another as morphisms. Categorical terminology is not, however, used and everything is explained from the bottom up again making it suitable for undergraduates.

The kernel Chapter, which the first four chapters merely serve to set up the machinery for, is Chapter 5. Fitting shows that EFS(string) can be implemented on an abstract model of a register machine. In the process of doing so he introduces two more programming languages IMP, a Pascal-like language and LOG a Prolog-like language and shows that they are all computationally equivalent and this he puts forward as empiric evidence for the Church-Turing thesis. The inspiration for the book can, perhaps, be seen in the paper by Tarnlund [4] on Horn clause computability.

The final Chapter is the Orobos, in which the worm eats its tail and programs are treated as data. Decidability is defined and the theorem that the halting problem is not decidable proved.

This book will not replace Lloyd as the main reference for the semantics of logic programming but I can't believe, as has been rumored, that this is the intention. The book is an undergraduate level text (in the UK at least) on computability from

the unfashionable viewpoint of recursively enumerable relations. To this end the book has been carefully executed with all details explained. Knowing my colleagues, I feel certain that it will not, unfortunately, find a sympathetic audience either in the Logic Programming camp because of its emphasis on computability or in the theory of computation camp because of its bias to logic programming. Those who were expecting a more careful treatment of the semantics of logic programming than is provided by Lloyd will have to wait for the reportedly forthcoming book by Lassez and Maher.

References

[1] J. Lloyd (1987, 1984) *Foundations of Logic Programming* (Springer, Berlin, 1987).
[2] J.-L. Lassez, M. J. Maher and K. Marriott, Unification revisited, Technical Report, IBM Yorktown Heights, 1987.
[3] J.-L. Lassez, and M. J. Maher, *The Semantics of Logic Programs* (Oxford University Press, Oxford).
[4] S. Tarnlund, Horn clause computability, *BIT* 17 (1977) 215-226.

Graem RINGWOOD
Department of Computing
Imperial College
London, United Kingdom

The Design and Evaluation of a High-Performance Smalltalk System. By David Michael Ungar. MIT Press, Cambridge, MA, 1987, Price £26.95.

Another book in the ACM distinguished dissertation series from MIT Press, Ungar's thesis describes the implementation of Smalltalk on a derivative of the RISC-2 processor at the University of California, Berkeley. The Smalltalk on a RISC (SOAR) project encompassed both software and hardware techniques for increasing the efficiency of Smalltalk execution to a level suitable for implementation on inexpensive processors. Smalltalk suffers the overheads of dynamic binding, run-time type checking, garbage collection and heap-allocated stacks. In addition to the interpretive overheads, this makes naïve Smalltalk implementations painfully slow. By compiling Smalltalk and providing the suitable architectural features, a SOAR processor with modest cycle time (400 ns) implements many macro-benchmarks faster than the definitive Dorado implementation (70 ns micro-cycle time).

Most of the performance gains were achieved by software techniques suitable for implementation on any machine. Although compilation is the most important element, it is only briefly discussed in the book and little information is given about the compiler. However, effective techniques such as in-line caching of call targets and direct addressing are investigated. The main thrust of software techniques was aimed at improving garbage collection which typically accounts for 10%-15% of execution time. The generation scavenging garbage collector is described in detail (pidgin C source included) and statistics gathered to illustrate the effectiveness of the scheme and highlight areas for improvement. Generation scavenging maintains two areas of objects—a "newspace" which is garbage collected frequently (every