This book provides an elementary introduction to classical recursive descent compiling and is aimed at the undergraduate. Its style is clear and unambiguous. The prerequisites of the reader are that they are familiar with elementary MODULA-2, basic data structures and a simple assembly language. MODULA-2 is used both as a source language and an implementation language to illustrate one compiling technique. The reader is not required to have a mathematical background and no knowledge of automata theory is required.

The author takes an evolutionary approach rather than top-down approach. This works well, introducing the reader to simple concepts before leading through to more complex ideas. The reader is introduced to the overall concept of compiling by a description of the main elements of a compiler, which sets the scene for the rest of the text. The book then continues with assembly language before introducing parsing.

The assembly language SAL—Simplified Assembly Language—(a simplified version of Intel 8086 assembly language) is used throughout the book to provide a practical illustration of the concepts of compiling. The assembler uses recursive descent and it is the use of the assembler rather than any abstract ideas which leads the student to understand recursive descent compiling. Readers need to understand basic SAL and its explanation in Chapter 2 is relatively straightforward.

Compiling is tackled by describing its components and then introducing the methods of compiling expressions before continuing on to compiling control structures through to compiling procedures and modules. Final chapters include error recovery and a brief introduction to compiler optimization. Each chapter concludes with a discussion (useful for student revision) and exercises.

The code used to illustrate concepts throughout the book is very readable. The absence of overflow checks, etc., avoids unnecessary distraction for the student trying to grasp the basic concepts. The software is available on disk from the publisher.

The strength of the text is its clear concise style with an undaunting approach. It is unashamedly aimed at the non-expert and assumes very little knowledge of the subject. Some may consider its weakness to be its lack of depth but the aim of the book is to provide an elementary grounding in the subject. Other methods of compiling are not referred to in the text and this may be considered a disadvantage. Certainly, a section containing a discussion of other types of compiler may be useful.

Overall, the book offers a straightforward explanation and illustration of recursive descent compiling for the undergraduate.

JUDITH JONES
De Montfort University

GEORGE COULOURIS, JEAN DOLLIMORE AND TIM KINDBERG
Distributed Systems—Concepts and Design. 2nd edn. Addison-Wesley. 1994. ISBN 0-201-62433-8. £23.95, 644 pp. hardbound.

The second edition of *Distributed Systems—Concepts and Design* is very different from the first: it has an additional author and it is more than twice as thick.

The book briefly puts distributed systems in a historical perspective and then covers networking, IPC and RPC, structure of distributed operating systems, file servers and name servers, time and coordination, replication, centralized and distributed transactions, concurrency control, recovery and fault tolerance, security, and distributed shared memory. The book ends with the presentation of several case studies.

The book reflects a preference of the authors for breadth rather than depth. It is very complete in its coverage of important experimental distributed-systems research, but the price for this is that theory of distributed systems is only marginally covered.

The preface suggests that the book can be used for undergraduate as well as for postgraduate teaching. Although the volume of the material in the book certainly justifies this (covering the whole book would take roughly 50 h of lecturing), I find the level of presentation basically that of undergraduate teaching. The book explains principles but not algorithms, it explains what but not how.

Chapter 3 ('Networking and Internetworking'), for example, presents the OSI reference model and explains what each layer is supposed to achieve, but does not discuss how this is done. I believe treatment of fault models and fault tolerance is essential in a book on distributed systems and I find the treatment of this subject too shallow. What I find especially missing is a discussion of the fundamental possibilities and impossibilities for masking faults under various fault models.

In spite of this defect, I find the book very useful as a first introduction to distributed systems. All the material on distributed systems I could ever hope to teach in an undergraduate course is present and up to date. The descriptions of the systems discussed are balanced and clear. Questions, at the end of each chapter, are useful for students to test their understanding of the material.

SAPE J. MULLENDER
University of Twente

ANDREAS PAEPCKE (ed.)
Object Oriented Programming: The CLOS Perspective. Cambridge University Press. 1993. ISBN 0 13 092990 5. £27.95. 202 pp. hardbound.

This is not a book about the methodology of object-oriented programming nor is it a book that teaches you how to program using the Common Lisp Object System (CLOS). Rather, it is a book that is intended to demonstrate the influence that the features of a particular object-oriented programming language, i.e. CLOS, can have on your approach to object-oriented programming. In other words, a book that provides a

view of object-oriented programming in general form a CLOS perspective.

The book takes the form of a collection of papers and is divided up into five parts. Each part has a brief introduction describing its purpose and the papers it contains. Part I Introduction opens with 'An introduction to CLOS' which concentrates on some of the more unusual features of the language and is intended to give a reader unfamiliar with CLOS enough background to understand the rest of the book. Although this introduction gets bogged down in obscure details about the semantics of multiple inheritance at one point, I think it succeeds in its goal overall (although I knew something about CLOS already). Unfortunately, to get the most out of the book you also need to have at least some awareness of Common Lisp syntax and semantics already, but this is probably unavoidable in a book of this nature and if you have got a rough idea about how Lisp works, you can follow what is going on most of the time.

If you did not know anything about CLOS before you opened the book, the introductory chapter is likely to challenge your view of object-oriented programming. CLOS takes a functional view of object-oriented programming which means that methods do not belong to any particular class and can be specialized on more than one argument. For example, the way a person eats something might depend on whether they're an adult or a child and also on what they're eating (ice-cream, jelly, biscuit, etc.). Most object-oriented languages would require you to decide whether the eat method belongs to the Person class or the Food class but in CLOS it belongs to neither and can be specialized on both.

Some of the philosophy behind CLOS and the reasons why the language is designed the way it is are described in the second paper in the Introduction, 'CLOS in Context—The Shape of the Design Space'. This is a very interesting paper for anyone interested in language design because it highlights a number of language design choices and discusses various trade-offs between them.

Part II contains a number of papers about the language feature for which CLOS is possibly most famous, its metaobject protocol. Although the basic architecture of CLOS is fixed (classes, methods, slots, etc.), the language is intended to be extensible. Thus, it is possible to define new kinds of classes which behave in different ways from a standard class, e.g. persistent classes might store their instances in a database whilst atomic classes might use locks to prevent concurrent method invocations on a particular instance. A metaobject protocol is a technique for opening up the implementation of a system and allowing it to be changed in a controlled but incremental fashion using the techniques of object-oriented programming at the meta-level (system or implementation level) rather than the application level. The four papers in this section present various examples of meta-level programming in CLOS and also discuss the use of metaobject protocols in a wider system context.

Part III of the book contains three papers comparing CLOS with other well-known object-oriented programming languages (C++, Eiffel and Sather, and Smalltalk). Although it might be expected that the more unusual features of CLOS would crop up again and again in each comparison, making the material rather repetitive, in fact this does not happen because each comparison is from a slightly different perspective. For example, the C++ comparison is language based and concentrates on the semantics of inheritance, the Eiffel/Sather comparison is more systematic and based on a framework of questions about different aspects of object-oriented programming systems, whilst the Smalltalk comparison concentrates on the meta-level and ends up proposing a new class model for Smalltalk.

Part IV presents three papers on CLOS uses and methodology. The first paper is about documenting object-oriented software but was too CLOS-specific and did not offer any new insights in my view. The second paper describes the implementation of LispView, an interface to Open-Look built using CLOS. It ends with a summary of lessons learned from using CLOS which argues that some of the more exotic CLOS language features really are indispensable. The final paper describs the use of CLOS to implement a hybrid knowledge representation tool. This application had stretched the metaobject protocol to its limits (and beyond!) so the paper contains an interesting critique of this aspect of CLOS.

Part V is concerned with the efficient implementation of CLOS on both specialized hardware (Lisp machines) and general-purpose architectures. The two papers in this section concentrate on the way in which method dispatch is implemented in CLOS but both are rather specialised and perhaps of less interest to the general reader than the rest of the book.

The book ends with a useful biography of each contributor, something which is often missing from a multi-author book of this nature.

Overall, I found the CLOS perspective on object-oriented programming presented here to be very interesting. The book contains a rich amount of material covering a wide range of topics. Although some of the material has appeared before in conference proceedings, this is still a useful collection to have if you are interested in CLOS and worth dipping into even if you are not. My only proviso would be that some familiarity with Lisp syntax is required in order to get the most out of the book but I do not consider this to be a major obstacle.

ROBERT STROUD
University of Newcastle

EDWARD V. BERARD
Essays on Object-Oriented Software Engineering: