both need a sufficient dose of evolving theory so as to improve it correspondingly through research, or through better engineering by deploying technology based on such theory.

As MODULA 2 is the chosen compiler implementation language, the presentation of the design and implementation exhibits better delineation of information pertinent to the different phases of compilation. This is a decided improvement over books which have presented similar designs expressed in PASCAL or C, as they do not have even first-level support for abstract data types. The use of MODULA 2 allows locality in modelling of information pertinent to different compilation phases and activities, such as symbol table definition and use, checks for well-formed MODEL program structures in semantic analysis, and code generation in the separate chapters on these topics. Integration effected through IMPORT of these modules is relatively clean, compared to the distribution of such information in older recursive descent based compilers for PASCAL (which led to a stepwise enrichment of phase-based information structures in all of the recursive-descent parsing procedures in these compilers).

An important part of the book is the collection of exercises at the end of each chapter. These exercises are themselves a source of learning, as they extend the discussion of compilation to aspects of programming languages that are not a part of MODEL.

The presentation in terms of content and style is a hop-step-and-jump from the structured programming movement that started over 25 years ago. The sprint for this event started with the author's involvement in porting Ammann's CDC 6600 Pascal Compiler (developed by stepwise refinement based on recursive descent parsing) to the ICL 1900 series, with two other books in this series co-authored by his senior collaborator, Jim Welsh, as the hop and the step. It is in this regard that the book falls short.

The proponents of structured programming have steadily moved on to increasingly formalized descriptions of the programming process. The directions they have set for themselves is to establish a science of programming. Taking this spirit forward requires that even complex programming problems, such as compiler construction, be addressed formally. Hence, there is no escape from attempting to use underlying programming language theory in any discussion on compiler construction, regardless of the current state-of-the-art of this theory, or the universal acceptance of any particular formalisation. Such a discussion shows up the scientific versus intuitive aspects of compilation. It is relative to their theory that the compilation function as a syntactic transformation under semantic invariance should be modelled. Such a discussion could also provide a basis for expressing engineering design concerns in the efficiency, robustness, and fault tolerance (error recovery!) in compilers. All these properties are clearly enjoyed by the developed compiler, for it presents a stable architecture that has evolved over time. Making this formally explicit is a responsibility of proponents of good programming practices.

Another point of note: by perpetuating an old style of handcrafting of programs, the programmers of the future will get rooted into history. Automatic generation of compiler components through use of compiler–compiler tools is widely accepted in practice and readily available in standard environments such as UNIX. Indeed, it is the most pragmatic fallout from improvements in the theory of programming languages. Expressing compiler design and construction through use of tools allow the designer to concentrate on programming language specifications, rather than on programming. After all, formal specification of programming languages is motherhood to several formal specification approaches. Positive experience in use of this style can only rub off on other programming exercises as well.

Till the arrival of the first solutions to retargetable code generation methods, the structure of compilers was completely dominated by the structure of source languages. Today, the idea that code generation can be modelled as a tree-pattern-matching based covering-problem is widely accepted. Systematic approaches to code selection can be explored—even through use of parsing technology. Locally optimal code selection is possible without resorting to complex flow analyses, all through modelling of target machines, and using tool-based approaches to deriving the code selection part of code generation. Posing some aspects of code generation to be local optimisation problems, and others (such as global register allocation or instruction scheduling for a pipe-lined RISC processor) as global optimisation problems, introduces a sharper focus on modelling of engineering properties of generated code.

The use of tool based approaches in teaching compiler construction provides a path for scale-up from classroom toy exercises to full-blown programming languages and industrially available target machines.

Today's compilers interface to a comprehensive programming environment, consisting of editors, browsers, debuggers and program composition facilities. Some of these aspects have been mildly dealt with in this book. The code generated by today's compilers also need to interface with a wide variety of extra-linguistic platforms, such as operating systems through run-time support libraries, database systems, graphical user interface management systems, etc. They also need to accommodate use of program fragments written in other programming languages. As pragmatic matters in compiler construction, they too need to be addressed.

KESAV V. NORI
Tata Research Development and Design Centre

JULIAN ULLMANN
Compiling in MODULA-2. Prentice Hall. 1994. ISBN 0-13-088741-2. £19.95. 425 pp. softbound.

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