2. Although several case studies are presented in the book, these are all of fairly small systems and it is not clear how easily the method employed could be adapted to work on large systems.
3. The specification language presented does not contain a construct analogous to what is known as a *module*, *package* or *class* in a programming language.

The book contains a large number of exercises, but neither are answers included nor is an instructor's manual available. Because of this it is unlikely that many people will think of using it as a textbook.

Having said all that, this book contains the first steps of a mathematical research programme whose development I will follow with interest. My main criticism of the book is not with its content, but with its manner of presentation. It is sloppy. It would have been a much better book had it been properly copy-edited and proof-read. I do not have space to list all the mistakes I found, but I will mention a sample of them:

- Set comprehensions are incorrectly typeset throughout the book. A thin amount of space should be added just after the opening brace and just before the closing one of each set comprehension.
- From the table of contents you would not know that the book contains an index and a bibliography.
- The three appendices repeat many definitions and propositions contained in the text. This is useful, but each definition and proposition so repeated is relabelled as well. For example, definition 1.3 on p. 18 becomes A.11 on p. 358. This is confusing—as is the fact that formula (x) on p. 18 has become (m) on p. 358 and the labels of the 11 following formulas have changed as well.
- I learnt that the book makes use of previously published material when I came across the words 'this paper' on p. 37. Usually such information is conveyed in the preface or the acknowledgements.
- Sometimes 'formulae' is used as the plural of 'formula' and sometimes 'formulas'.
- On p. 64 a relation is defined as being well-founded if and only if it is transitive and progressively finite, whereas on p. 356 a well-founded relation is defined as one whose *inverse* is transitive and progressively finite.
- Sometimes angle brackets are used to represent ordered pairs and sometimes parentheses. If there is a difference between these, it is not explained.
- The notation for the power of a relation is used from p. 9 onwards, but is only defined on p. 63.

Some of these may seem fairly minor mistakes, but this book contains so many of them that they constitute a major problem in that the task of understanding the material being presented is much harder than it should be.

ANTONI DILLER
University of Birmingham

DAVID A. WATT
Programming Language Processors. Prentice-Hall International. 1993. ISBN 0-13-720129-X. £19.95. 452 pp. softbound.

Amongst books on compilers in this series that include extensive case studies, I like this one most—and this despite the fact that its case studies present a compiler for a toy source language generating code for a hypothetical target machine, as well as an interpreter for the same source. The main reason for this bias is its integration of formal semantics and semantic preservation in its discussion on programming language processors (PLPs). Most text books on compilers are remiss on this account.

For best appreciation, this book has to be read in the context of its predecessors by the same author: *Programming Language Concepts and Paradigms* and *Programming Language Syntax and Semantics*. From concepts to definitions, and from there to the design and implementation concerns, is a progress which empathizes with the programming process. Even if the reader takes the direct path to this book, as this reviewer did, Chapter 1 introduces in brief the three-fold concerns in the definition of programming languages: context-free syntax, contextual staticsemantics, and dynamic behavioural-semantics. The author's choices of definitional mechanisms are EBNF for concrete syntax, procedural mechanisms to decorate abstract syntax trees for contextual semantic-information and Mosses' Action-Semantics for behavioural-semantics of programs. With these as the foundation, the discussion on PLPs commences.

Chapter 2 discusses translators, compilers and interpreters as important examples of programming language processors. It uses tombstone diagrams, a refinement of T-diagrams, to describe lucidly the nature of PLPs as programs. It also discusses portability and bootstrapping of compilers in clear terms, something usually missing from introductory books.

Chapter 3 introduces the overall separation of technical concerns in compiler design, which governs the structure of subsequent chapters. Chapter 4 deals with syntax analysis, Chapter 5 with contextual analysis, and Chapters 6 and 7 deal with run-time storage-organisation and code-generation respectively. Chapters 4, 5, 6 and 8 on interpretation can be used to understand the construction of interpreters. The appendices collect definitions and code for the case studies. All chapters contain exercises at three levels of difficulty, ranging from quiz questions to complete projects. An appendix also gives the author's answers to some of the exercises.

To this reviewer, these exercises, as well as the detailed use of action-semantics and semantic equations to express semantic-equivalence concerns when designing run-time storage-organisation and code-generation patterns, are the most valuable part of the book. There are simple arguments for asserting the correctness of

Downloaded from https://academic.oup.com/comjnl/article/38/9/751/363072 by guest on 10 April 2024

generated code and its execution environment. The correctness of syntax analysis and contextual analysis is less formal and is principally achieved by pointing to structural similarities between corresponding definitions and compiler-phase-code as a reason to believe its correctness. This part of the specification of compilation has a procedural flavour.

Retargetable code-generation is one of the important advances in compilation by-passed in this discussion on compilation. Even at the introductory level, it is important to include this idea because it needs a conscious shift from the source-language centered view of the compilation process. Whilst the importance of the source-language-centered view cannot be denied in the analysis of source programs, it is not a good method for perceiving locality in generated target code, causing many redundancies. Choosing well-conceived hypothetical machines whose architecture intrinsically supports the architectural aspects of the source language is fine for devising bootstrapping compilers. In this special circumstance, code generation is comfortably couched within the folds of a recursive-descent parser. Even some simple optimizations such as avoiding redundant LOAD/STORE operations, folding of constant expressions, or even jump-chain elimination can be cleanly expressed, despite the fact that these aspects of generated code cross syntatic boundaries that are procedurally encapsulated in a recursive descent parser. (On another track, table-driven approaches are far better at error-handling and repair than the control-flow based methods of handling analysis through recursive-descent, common to many books in this series.)

This approach does, however, hide the real-life problems of devising good code generators for real-life target machines. Typical code generation concerns that are hidden by such a treatment are the following: language architecture mapping through reservation of registers and use of memory system architecture, data storage layout and anticipated accesses mapped onto effective address computation mechanisms in the instruction set architecture, uses of registers and a (simulated) stack for managing temporary intermediate values during computation, and global resource management.

Good retargetable-code-generation methods make it possible to formulate cleanly most of these problems by focusing on the description of the target machine. The use of generic code-generation algorithms make it possible to teach code generation cleanly without hiding real-life issues. They even allow the formulation and solution of local-optimality of coding, a step in the direction of quantitative issues underlying the discipline of programming dictated by good engineering concerns, over and above the important semantic concerns central to this book. It also systematizes an insight into the supposedly arcane practices of assembly language programmers of bygone days.

In moving towards some formalization as a basis for development of compilers, this book is a step forward. Nevertheless, the book continues the hand-crafting tradition of programming, a stumbling block to programmer productivity. Over 30 years of compiler-compiler research have uncovered some gems which overcome some of the productivity bottlenecks in a manner that is simultaneously elegant and efficient, rendering parts of compiler construction to be a tool-based program construction activity. Ignoring these developments is not progress. Waiting for the ultimate means of specification methods from which program construction code could be derived is not engineering pragmatism. Programming is also an engineering discipline, not only a formal and mathematical area of study. Compiler construction exists because it is an exercise in engineering a desirable virtual computer. For all these reasons, correctness concerns notwithstanding, teaching about programming language processors provides a test-bed for evolving an engineering discipline for programs and programming. It is a responsibility which cannot be passed by.

Today's computer science and software engineering students are bred on workstation and personal computer culture. They are fully conversant with program development tools (sometimes to the chagrin of an older generation of teachers who learnt their trade on punched-card machines, as in the case of this reviewer). The transition to tool-based approaches should not be difficult. Moreover, tool-based approaches facilitate portability in the same way as the author's hypothetical computers and their (small) interpreters do.

Finally, some personal notes. It is time to move on from Pascal. Even Wirth has. In sticking to languages which can be cleanly dealt with, the gap between the well-understood-and-taught and practitioner's reality becomes emphasized. Defining good abstract machines is a good way of capturing the essence of different paradigms of program organisation and computation. Several already exist, such as FAM and WAM for functional and logic programming. (FAM is perhaps dated; there must be similar developments underlying Haskell or Gofer.) The ObjVLisp's Object Model gives a good handle on working with inheritance. Modula-2's machine-oriented approach to concurrency through co-routines is another mechanistic handle. These good and clean ideas at the abstract machine level are needed when considering implementation through compilation. The trilogy approach of the author provides a good divide-and-conquer strategy for finding unity in this diversity. This has been the experience of the reviewer in experimenting with teaching programming languages and their processors over the past 20 years, though not in this explicit form in the initial 5 years.

KESAV V. NORI
Tata Research Development and Design Centre
Pune, India

Downloaded from https://academic.oup.com/comjnl/article/38/9/751/363072 by guest on 10 April 2024