rithmic Logic, introduced by A. Salwicki and G. Mirkowska also and perhaps better known as Harel's dynamic logic, four constructs are added to the assertion language: initially assertions, always assertions, sometimes assertions and previously assertions, whereas the latter ones may only occur in always assertions and previously assertions. This additional constructs offer the full power of dynamic logic for the specification of procedures.

COLD-K is not only a specification language it is also a *design* language that can be used in all intermediate stages of design ranging from specification to implementation. This aim is achieved by constructs in COLD-K that allow the description of algorithms including the concept of side-effects in case of state-based algorithms.

The main goal of the book is not the description of the language features but to show how algebraic and state-based specification techniques can be effectively used in the software development process. This aim is reflected in the structure of the book. In Part I algebraic specification techniques are presented together with methodological guidelines about structuring and implementing algebraic specifications. Part II presents state-based specification techniques together with methodological guidelines about structuring and implementing state-based specifications. Part III contains some theoretical topics and advanced language features and it deals briefly with the specification of large systems. The language constructs and the methodological guidelines are presented together with instructive examples. Especially the methodological topics make the present book very recommendable.

Since the present book is not the only one and even not the first one on the market about formal specification and design there should be a serious comparison with similar languages and systems, for instance with the RSL language and the RAISE system.

The comparison of COLD-K with the initial algebra approach in Section 9.3 is not well-founded. The described advantages of COLD-K are only correct in the case of algebraic specifications that are completely initially constraint. Most languages based on the initial algebra approach offer the possibility of initial constraints as axioms in addition to conditional equations. In that case the described advantages of COLD-K disappear.

HORST REICHEL
Institute of Theoretical Computer Science
Faculty of Computer Science
TU Dresden
D-01062 Dresden
Germany

F. Nielson and H.R. Nielson, *Two-Level Functional Languages*, Cambridge Tracts in Computer Science, Vol. 34 (Cambridge University Press, 1992)

This book is an excellent exposition of the research by the authors (and others) in the area of *semantic techniques* for the efficient and correct implementation of (functional)

programming languages. It is particularly suitable for researchers and postgraduate students. Some knowledge of Typed Calculi and Denotational Semantics is an essential prerequisite.

The book provides a rational reconstruction of several techniques (scattered in the literature) and places them in a coherent and unifying context. To exemplify these techniques, the authors apply them to a lazy functional language, which amounts to a lambda-calculus with lists.

In this subject proofs tend to be rather long and boring, with a lot of induction and case analysis, but the authors have found a reasonable compromise. They assist the reader who wants to go through all steps, by breaking long proof into a sequence of (meaningful) lemmas, by commenting definitions and also critical steps in a proof, by providing several indexes and a list of tables.

Each chapter includes bibliographic notes, with pointers to the literature, and exercises, either to fill the gaps or to propose variations on the theme of the chapter.

The first part of the book (Chapters 2–4) is devoted to *making concepts explicit*. The general idea is that before generating (optimized) code corresponding to a high-level program, the program should undergo some massaging. Chapter 2 introduces explicit type information, the techniques are standard, and it is a warming up for the following chapter. Chapter 3 introduces the distinction between compile-time and run-time, the attribute two-level in the title refers to this distinction. Chapter 4 takes the distinction between the two levels to the extreme, by differentiating the syntax for compile-time and run-time expressions: lambda-terms for the former, combinators for the latter.

Chapter 5 exploits the fully explicit language, obtained via the previous massaging, to introduce a flexible notion of semantics: parametrized semantics. In this semantics the interpretation of compile-time expressions is parametric in an interpretation of run-time expressions. By choosing different interpretations for the run-time expressions, one can capture in this framework several semantics: the *standard* denotational semantics, various forms of abstract interpretation, and (optimized) code generation.

The following chapters (Chapter 6–7) consider specific instances of parametrized semantics and prove their *correctness* w.r.t. the standard semantics, using the technique of (Kripke) logical relations.

In the last chapter the authors hint to other parametrized semantics, combining abstract interpretation and code generation, for generating optimized code. They discuss also ways of adapting the ideas developed in a purely functional context to the implementation of an imperative language.

E. MOGGI
DISI
Università di Genova
viale Benedetto XV 3
16132 Genova
Italy