Preface

The first programming language I ever learned was ALGOL60. This language was notable for its elegance and its regularity; for all its imperfections, it stood head and shoulders above its contemporaries. My interest in languages was awakened, and I began to perceive the benefits of simplicity and consistency in language design.

Since then I have learned and programmed in about a dozen other languages, and I have struck a nodding acquaintance with many more. Like many programmers, I have found that certain languages make programming distasteful, a drudgery; others make programming enjoyable, even esthetically pleasing. A good language, like a good mathematical notation, helps us to formulate and communicate ideas clearly. My personal favorites have been PASCAL, ADA, ML, and JAVA. Each of these languages has sharpened my understanding of what programming is (or should be) all about. PASCAL taught me structured programming and data types. ADA taught me data abstraction, exception handling, and large-scale programming. ML taught me functional programming and parametric polymorphism. JAVA taught me object-oriented programming and inclusion polymorphism. I had previously met all of these concepts, and understood them in principle, but I did not *truly* understand them until I had the opportunity to program in languages that exposed them clearly.

Contents

This book consists of five parts.

Chapter 1 introduces the book with an overview of programming linguistics (the study of programming languages) and a brief history of programming and scripting languages.

Chapters 2–5 explain the basic concepts that underlie almost all programming languages: values and types, variables and storage, bindings and scope, procedures and parameters. The emphasis in these chapters is on identifying the basic concepts and studying them individually. These basic concepts are found in almost all languages.

Chapters 6–10 continue this theme by examining some more advanced concepts: data abstraction (packages, abstract types, and classes), generic abstraction (or templates), type systems (inclusion polymorphism, parametric polymorphism, overloading, and type conversions), sequencers (including exceptions), and concurrency (primitives, conditional critical regions, monitors, and rendezvous). These more advanced concepts are found in the more modern languages.

Chapters 11–16 survey the most important programming paradigms, comparing and contrasting the long-established paradigm of imperative programming with the increasingly important paradigms of object-oriented and concurrent programming, the more specialized paradigms of functional and logic programming, and the paradigm of scripting. These different paradigms are based on different

selections of key concepts, and give rise to sharply contrasting styles of language and of programming. Each chapter identifies the key concepts of the subject paradigm, and presents an overview of one or more major languages, showing how concepts were selected and combined when the language was designed. Several designs and implementations of a simple spellchecker are presented to illustrate the pragmatics of programming in all of the major languages.

Chapters 17 and 18 conclude the book by looking at two issues: how to select a suitable language for a software development project, and how to design a new language.

The book need not be read sequentially. Chapters 1–5 should certainly be read first, but the remaining chapters could be read in many different orders. Chapters 11–15 are largely self-contained; my recommendation is to read at least some of them after Chapters 1–5, in order to gain some insight into how major languages have been designed. Figure P.1 summarizes the dependencies between the chapters.

Examples and case studies

The concepts studied in Chapters 2–10 are freely illustrated by examples. These examples are drawn primarily from C, C++, JAVA, and ADA. I have chosen these languages because they are well known, they contrast well, and even their flaws are instructive!

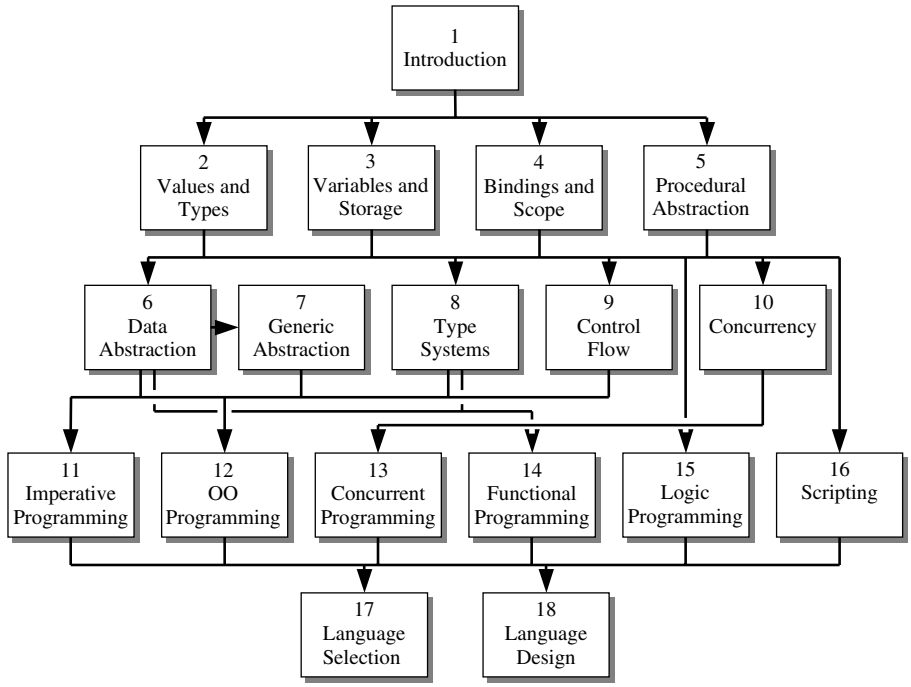

Figure P.1 Dependencies between chapters of this book.

The paradigms studied in Chapters 11–16 are illustrated by case studies of major languages: Ada, C, C++, Haskell, Java, Prolog, and Python. These languages are studied only impressionistically. It would certainly be valuable for readers to learn to program in all of these languages, in order to gain deeper insight, but this book makes no attempt to teach programming *per se*. The bibliography contains suggested reading on all of these languages.

Exercises

Each chapter is followed by a number of relevant exercises. These vary from short exercises, through longer ones (marked *), up to truly demanding ones (marked **) that could be treated as projects.

A typical exercise is to analyze some aspect of a favorite language, in the same way that various languages are analyzed in the text. Exercises like this are designed to deepen readers' understanding of languages that they already know, and to reinforce understanding of particular concepts by studying how they are supported by different languages.

A typical project is to design some extension or modification to an existing language. I should emphasize that language design should not be undertaken lightly! These projects are aimed particularly at the most ambitious readers, but all readers would benefit by at least thinking about the issues raised.

Readership

All programmers, not just language specialists, need a thorough understanding of language concepts. This is because programming languages are our most fundamental tools. They influence the very way we think about software design and implementation, about algorithms and data structures.

This book is aimed at junior, senior, and graduate students of computer science and information technology, all of whom need some understanding of the fundamentals of programming languages. The book should also be of interest to professional software engineers, especially project leaders responsible for language evaluation and selection, designers and implementers of language processors, and designers of new languages and of extensions to existing languages.

To derive maximum benefit from this book, the reader should be able to program in at least two contrasting high-level languages. Language concepts can best be understood by comparing how they are supported by different languages. A reader who knows only a language like C, C++, or Java should learn a contrasting language such as Ada (or *vice versa*) at the same time as studying this book.

The reader will also need to be comfortable with some elementary concepts from discrete mathematics – sets, functions, relations, and predicate logic – as these are used to explain a variety of language concepts. The relevant mathematical concepts are briefly reviewed in Chapters 2 and 15, in order to keep this book reasonably self-contained.

This book attempts to cover all the most important aspects of a large subject. Where necessary, depth has been sacrificed for breadth. Thus the really serious

student will need to follow up with more advanced studies. The book has an extensive bibliography, and each chapter closes with suggestions for further reading on the topics covered by the chapter.

Acknowledgments

Bob Tennent's classic book *Programming Language Principles* has profoundly influenced the way I have organized this book. Many books on programming languages have tended to be *syntax-oriented*, examining several popular languages feature by feature, without offering much insight into the underlying concepts or how future languages might be designed. Some books are *implementation-oriented*, attempting to explain concepts by showing how they are implemented on computers. By contrast, Tennent's book is *semantics-oriented*, first identifying and explaining powerful and general semantic concepts, and only then analyzing particular languages in terms of these concepts. In this book I have adopted Tennent's semantics-oriented approach, but placing far more emphasis on concepts that have become more prominent in the intervening two decades.

I have also been strongly influenced, in many different ways, by the work of Malcolm Atkinson, Peter Buneman, Luca Cardelli, Frank DeRemer, Edsger Dijkstra, Tony Hoare, Jean Ichbiah, John Hughes, Mehdi Jazayeri, Bill Joy, Robin Milner, Peter Mosses, Simon Peyton Jones, Phil Wadler, and Niklaus Wirth.

I wish to thank Bill Findlay for the two chapters (Chapters 10 and 13) he has contributed to this book. His expertise on concurrent programming has made this book broader in scope than I could have made it myself. His numerous suggestions for my own chapters have been challenging and insightful.

Last but not least, I would like to thank the Wiley reviewers for their constructive criticisms, and to acknowledge the assistance of the Wiley editorial staff led by Gaynor Redvers-Mutton.

David A. Watt
Brisbane
March 2004