Concepts, Techniques, and Models of Computer Programming

PETER VAN ROY[1]
Université catholique de Louvain (at Louvain-la-Neuve)
Swedish Institute of Computer Science

SEIF HARIDI[2]
Royal Institute of Technology (KTH)
Swedish Institute of Computer Science

June 5, 2003

[1]Email: pvr@info.ucl.ac.be, Web: http://www.info.ucl.ac.be/~pvr
[2]Email: seif@it.kth.se, Web: http://www.it.kth.se/~seif

Preface

Six blind sages were shown an elephant and met to discuss their experience. "It's wonderful," said the first, "an elephant is like a rope: slender and flexible." "No, no, not at all," said the second, "an elephant is like a tree: sturdily planted on the ground." "Marvelous," said the third, "an elephant is like a wall." "Incredible," said the fourth, "an elephant is a tube filled with water." "What a strange piecemeal beast this is," said the fifth. "Strange indeed," said the sixth, "but there must be some underlying harmony. Let us investigate the matter further."
– Freely adapted from a traditional Indian fable.

"A programming language is like a natural, human language in that it favors certain metaphors, images, and ways of thinking."
– Mindstorms: Children, Computers, and Powerful Ideas [141], *Seymour Papert* (1980)

One approach to study computer programming is to study programming languages. But there are a tremendously large number of languages, so large that it is impractical to study them all. How can we tackle this immensity? We could pick a small number of languages that are representative of different programming paradigms. But this gives little insight into programming as a unified discipline. This book uses another approach.

We focus on programming *concepts* and the *techniques* to use them, not on programming languages. The concepts are organized in terms of computation models. A computation model is a formal system that defines how computations are done. There are many ways to define computation models. Since this book is intended to be practical, it is important that the computation model should be directly useful to the programmer. We will therefore define it in terms of concepts that are important to programmers: data types, operations, and a programming language. The term computation model makes precise the imprecise notion of "programming paradigm". The rest of the book talks about computation models and not programming paradigms. Sometimes we will use the phrase programming model. This refers to what the programmer needs: the programming techniques and design principles made possible by the computation model.

Each computation model has its own set of techniques for programming and reasoning about programs. The number of different computation models that are known to be useful is much smaller than the number of programming languages. This book covers many well-known models as well as some less-known models. The main criterium for presenting a model is whether it is useful in practice.

Each computation model is based on a simple core language called its *kernel language*. The kernel languages are introduced in a progressive way, by adding concepts one by one. This lets us show the deep relationships between the different models. Often, just adding one new concept makes a world of difference in programming. For example, adding destructive assignment (explicit state) to functional programming allows us to do object-oriented programming.

When stepping from one model to the next, how do we decide on what concepts to add? We will touch on this question many times in the book. The main criterium is the *creative extension principle*. Roughly, a new concept is added when programs become complicated for technical reasons unrelated to the problem being solved. Adding a concept to the kernel language can keep programs simple, if the concept is chosen carefully. This is explained further in Appendix D. This principle underlies the progression of kernel languages presented in the book.

A nice property of the kernel language approach is that it lets us use different models together in the same program. This is usually called *multiparadigm programming*. It is quite natural, since it means simply to use the right concepts for the problem, independent of what computation model they originate from. Multiparadigm programming is an old idea. For example, the designers of Lisp and Scheme have long advocated a similar view. However, this book applies it in a much broader and deeper way than was previously done.

From the vantage point of computation models, the book also sheds new light on important problems in informatics. We present three such areas, namely graphical user interface design, robust distributed programming, and constraint programming. We show how the judicious combined use of several computation models can help solve some of the problems of these areas.

Languages mentioned

We mention many programming languages in the book and relate them to particular computation models. For example, Java and Smalltalk are based on an object-oriented model. Haskell and Standard ML are based on a functional model. Prolog and Mercury are based on a logic model. Not all interesting languages can be so classified. We mention some other languages for their own merits. For example, Lisp and Scheme pioneered many of the concepts presented here. Erlang is functional, inherently concurrent, and supports fault tolerant distributed programming.

We single out four languages as representatives of important computation models: Erlang, Haskell, Java, and Prolog. We identify the computation model of each language in terms of the book's uniform framework. For more information about them we refer readers to other books. Because of space limitations, we are not able to mention all interesting languages. Omission of a language does not imply any kind of value judgement.

Goals of the book

Teaching programming

The main goal of the book is to teach programming as a unified discipline with a scientific foundation that is useful to the practicing programmer. Let us look closer at what this means.

What is programming?

We define *programming*, as a general human activity, to mean the act of extending or changing a system's functionality. Programming is a widespread activity that is done both by nonspecialists (e.g., consumers who change the settings of their alarm clock or cellular phone) and specialists (computer programmers, the audience of this book).

This book focuses on the construction of software systems. In that setting, programming is the step between the system's specification and a running program that implements it. The step consists in designing the program's architecture and abstractions and coding them into a programming language. This is a broad view, perhaps broader than the usual connotation attached to the word programming. It covers both programming "in the small" and "in the large". It covers both (language-independent) architectural issues and (language-dependent) coding issues. It is based more on concepts and their use rather than on any one programming language. We find that this general view is natural for teaching programming. It allows to look at many issues in a way unbiased by limitations of any particular language or design methodology. When used in a specific situation, the general view is adapted to the tools used, taking account their abilities and limitations.

Both science and technology

Programming as defined above has two essential parts: a technology and its scientific foundation. The technology consists of tools, practical techniques, and standards, allowing us to *do* programming. The science consists of a broad and deep theory with predictive power, allowing us to *understand* programming. Ideally, the science should explain the technology in a way that is as direct and useful as possible.

If either part is left out, we are no longer doing programming. Without the technology, we are doing pure mathematics. Without the science, we are doing a craft, i.e., we lack deep understanding. Teaching programming correctly therefore means teaching both the technology (current tools) and the science (fundamental concepts). Knowing the tools prepares the student for the present. Knowing the concepts prepares the student for future developments.

More than a craft

Despite many efforts to introduce a scientific foundation, programming is almost always taught as a craft. It is usually taught in the context of one (or a few) programming languages (e.g., Java, complemented with Haskell, Scheme, or Prolog). The historical accidents of the particular languages chosen are interwoven together so closely with the fundamental concepts that the two cannot be separated. There is a confusion between tools and concepts. What's more, different schools of thought have developed, based on different ways of viewing programming, called "paradigms": object-oriented, logic, functional, etc. Each school of thought has its own science. The unity of programming as a single discipline has been lost.

Teaching programming in this fashion is like having separate schools of bridge building: one school teaches how to build wooden bridges and another school teaches how to build iron bridges. Graduates of either school would implicitly consider the restriction to wood or iron as fundamental and would not think of using wood and iron together.

The result is that programs suffer from poor design. We give an example based on Java, but the problem exists in all existing languages to some degree. Concurrency in Java is complex to use and expensive in computational resources. Because of these difficulties, Java-taught programmers conclude that concurrency is a fundamentally complex and expensive concept. Program specifications are designed around the difficulties, often in a contorted way. But these difficulties are not fundamental at all. There are forms of concurrency that are quite useful and yet as easy to program with as sequential programs (for example, stream programming as exemplified by Unix pipes). Furthermore, it is possible to implement threads, the basic unit of concurrency, almost as cheaply as procedure calls. If the programmer were taught about concurrency in the correct way, then he or she would be able to specify for and program in systems without concurrency restrictions (including improved versions of Java).

The kernel language approach

Practical programming languages scale up to programs of millions of lines of code. They provide a rich set of abstractions and syntax. How can we separate the languages' fundamental concepts, which underlie their success, from their historical accidents? The kernel language approach shows one way. In this approach, a practical language is translated into a *kernel language* that consists of a small number of *programmer-significant* elements. The rich set of abstractions and syntax is encoded into the small kernel language. This gives both programmer and student a clear insight into what the language does. The kernel language has a simple formal semantics that allows reasoning about program correctness and complexity. This gives a solid foundation to the programmer's intuition and the programming techniques built on top of it.

A wide variety of languages and programming paradigms can be modeled by a small set of closely-related kernel languages. It follows that the kernel language approach is a truly language-independent way to study programming. Since any given language translates into a kernel language that is a subset of a larger, more complete kernel language, the underlying unity of programming is regained.

Reducing a complex phenomenon to its primitive elements is characteristic of the scientific method. It is a successful approach that is used in all the exact sciences. It gives a deep understanding that has predictive power. For example, structural science lets one design *all* bridges (whether made of wood, iron, both, or anything else) and predict their behavior in terms of simple concepts such as force, energy, stress, and strain, and the laws they obey [62].

Comparison with other approaches

Let us compare the kernel language approach with three other ways to give programming a broad scientific basis:

- A *foundational calculus*, like the λ calculus or π calculus, reduces programming to a minimal number of elements. The elements are chosen to simplify mathematical analysis, not to aid programmer intuition. This helps theoreticians, but is not particularly useful to practicing programmers. Foundational calculi are useful for studying the fundamental properties and limits of programming a computer, not for writing or reasoning about general applications.

- A *virtual machine* defines a language in terms of an implementation on an idealized machine. A virtual machine gives a kind of operational semantics, with concepts that are close to hardware. This is useful for designing computers, implementing languages, or doing simulations. It is not useful for reasoning about programs and their abstractions.

- A *multiparadigm language* is a language that encompasses several programming paradigms. For example, Scheme is both functional and imperative ([38]) and Leda has elements that are functional, object-oriented, and logical ([27]). The usefulness of a multiparadigm language depends on how well the different paradigms are integrated.

The kernel language approach combines features of all these approaches. A well-designed kernel language covers a wide range of concepts, like a well-designed multiparadigm language. If the concepts are independent, then the kernel language can be given a simple formal semantics, like a foundational calculus. Finally, the formal semantics can be a virtual machine at a high level of abstraction. This makes it easy for programmers to reason about programs.

Designing abstractions

The second goal of the book is to teach how to design programming abstractions. The most difficult work of programmers, and also the most rewarding, is not writing programs but rather *designing abstractions*. Programming a computer is primarily designing and using abstractions to achieve new goals. We define an *abstraction* loosely as a tool or device that solves a particular problem. Usually the same abstraction can be used to solve many different problems. This versatility is one of the key properties of abstractions.

Abstractions are so deeply part of our daily life that we often forget about them. Some typical abstractions are books, chairs, screwdrivers, and automobiles.[1] Abstractions can be classified into a hierarchy depending on how specialized they are (e.g., "pencil" is more specialized than "writing instrument", but both are abstractions).

Abstractions are particularly numerous inside computer systems. Modern computers are highly complex systems consisting of hardware, operating system, middleware, and application layers, each of which is based on the work of thousands of people over several decades. They contain an enormous number of abstractions, working together in a highly organized manner.

Designing abstractions is not always easy. It can be a long and painful process, as different approaches are tried, discarded, and improved. But the rewards are very great. It is not too much of an exaggeration to say that civilization is built on successful abstractions [134]. New ones are being designed every day. Some ancient ones, like the wheel and the arch, are still with us. Some modern ones, like the cellular phone, quickly become part of our daily life.

We use the following approach to achieve the second goal. We start with programming concepts, which are the raw materials for building abstractions. We introduce most of the relevant concepts known today, in particular lexical scoping, higher-order programming, compositionality, encapsulation, concurrency, exceptions, lazy execution, security, explicit state, inheritance, and nondeterministic choice. For each concept, we give techniques for building abstractions with it. We give many examples of sequential, concurrent, and distributed abstractions. We give some general laws for building abstractions. Many of these general laws have counterparts in other applied sciences, so that books like [69], [55], and [62] can be an inspiration to programmers.

Main features

Pedagogical approach

There are two complementary approaches to teaching programming as a rigorous discipline:

[1]Also, pencils, nuts and bolts, wires, transistors, corporations, songs, and differential equations. They do not have to be material entities!

- The *computation-based approach* presents programming as a way to define executions on machines. It grounds the student's intuition in the real world by means of actual executions on real systems. This is especially effective with an interactive system: the student can create program fragments and immediately see what they do. Reducing the time between thinking "what if" and seeing the result is an enormous aid to understanding. Precision is not sacrificed, since the formal semantics of a program can be given in terms of an abstract machine.

- The *logic-based approach* presents programming as a branch of mathematical logic. Logic does not speak of execution but of program properties, which is a higher level of abstraction. Programs are mathematical constructions that obey logical laws. The formal semantics of a program is given in terms of a mathematical logic. Reasoning is done with logical assertions. The logic-based approach is harder for students to grasp yet it is essential for defining precise specifications of what programs do.

Like *Structure and Interpretation of Computer Programs*, by Abelson, Sussman, & Sussman [1, 2], our book mostly uses the computation-based approach. Concepts are illustrated with program fragments that can be run interactively on an accompanying software package, the Mozart Programming System [129]. Programs are constructed with a building-block approach, bringing together basic concepts to build more complex ones. A small amount of logical reasoning is introduced in later chapters, e.g., for defining specifications and for using invariants to reason about programs with state.

Formalism used

This book uses a single formalism for presenting all computation models and programs, namely the Oz language and its computation model. To be precise, the computation models of this book are all carefully-chosen subsets of Oz. Why did we choose Oz? The main reason is that it supports the kernel language approach well. Another reason is the existence of the Mozart Programming System.

Panorama of computation models

This book presents a broad overview of many of the most useful computation models. The models are designed not just with formal simplicity in mind (although it is important), but on the basis of how a programmer can express himself/herself and reason within the model. There are many different practical computation models, with different levels of expressiveness, different programming techniques, and different ways of reasoning about them. We find that each model has its domain of application. This book explains many of these models, how they are related, how to program in them, and how to combine them to greatest advantage.

More is not better (or worse), just different

All computation models have their place. It is not true that models with more concepts are better or worse. This is because a new concept is like a two-edged sword. Adding a concept to a computation model introduces new forms of expression, making some programs simpler, but it also makes reasoning about programs harder. For example, by adding *explicit state* (mutable variables) to a functional programming model we can express the full range of object-oriented programming techniques. However, reasoning about object-oriented programs is harder than reasoning about functional programs. Functional programming is about calculating values with mathematical functions. Neither the values nor the functions change over time. Explicit state is one way to model things that change over time: it provides a container whose content can be updated. The very power of this concept makes it harder to reason about.

The importance of using models together

Each computation model was originally designed to be used in isolation. It might therefore seem like an aberration to use several of them together in the same program. We find that this is not at all the case. This is because models are not just monolithic blocks with nothing in common. On the contrary, they have much in common. For example, the differences between declarative & imperative models and concurrent & sequential models are very small compared to what they have in common. Because of this, it is easy to use several models together.

But even though it is technically possible, why would one *want* to use several models in the same program? The deep answer to this question is simple: because one does not program with models, but with programming concepts and ways to combine them. Depending on which concepts one uses, it is possible to consider that one is programming in a particular model. The model appears as a kind of epiphenomenon. Certain things become easy, other things become harder, and reasoning about the program is done in a particular way. It is quite natural for a well-written program to use different models. At this early point this answer may seem cryptic. It will become clear later in the book.

An important principle we will see in this book is that concepts traditionally associated with one model can be used to great effect in more general models. For example, the concepts of lexical scoping and higher-order programming, which are usually associated with functional programming, are useful in all models. This is well-known in the functional programming community. Functional languages have long been extended with explicit state (e.g., Scheme [38] and Standard ML [126, 192]) and more recently with concurrency (e.g., Concurrent ML [158] and Concurrent Haskell [149, 147]).

The limits of single models

We find that a good programming style requires using programming concepts that are usually associated with different computation models. Languages that implement just one computation model make this difficult:

- Object-oriented languages encourage the overuse of state and inheritance. Objects are stateful by default. While this seems simple and intuitive, it actually complicates programming, e.g., it makes concurrency difficult (see Section 8.2). Design patterns, which define a common terminology for describing good programming techniques, are usually explained in terms of inheritance [58]. In many cases, simpler higher-order programming techniques would suffice (see Section 7.4.7). In addition, inheritance is often misused. For example, object-oriented graphical user interfaces often recommend using inheritance to extend generic widget classes with application-specific functionality (e.g., in the Swing components for Java). This is counter to separation of concerns.

- Functional languages encourage the overuse of higher-order programming. Typical examples are monads and currying. Monads are used to encode state by threading it throughout the program. This makes programs more intricate but does not achieve the modularity properties of true explicit state (see Section 4.7). Currying lets you apply a function partially by giving only some of its arguments. This returns a new function that expects the remaining arguments. The function body will not execute until all arguments are there. The flipside is that it is not clear by inspection whether the function has all its arguments or is still curried ("waiting" for the rest).

- Logic languages in the Prolog tradition encourage the overuse of Horn clause syntax and search. These languages define all programs as collections of Horn clauses, which resemble simple logical axioms in an "if-then" style. Many algorithms are obfuscated when written in this style. Backtracking-based search must always be used even though it is almost never needed (see [196]).

These examples are to some extent subjective; it is difficult to be completely objective regarding good programming style and language expressiveness. Therefore they should not be read as passing any judgement on these models. Rather, they are hints that none of these models is a panacea when used alone. Each model is well-adapted to some problems but less to others. This book tries to present a balanced approach, sometimes using a single model in isolation but not shying away from using several models together when it is appropriate.

Teaching from the book

We explain how the book fits in an informatics curriculum and what courses can be taught with it. By *informatics* we mean the whole field of information technology, including computer science, computer engineering, and information systems. Informatics is sometimes called *computing*.

Role in informatics curriculum

Let us consider the discipline of programming independent of any other domain in informatics. In our experience, it divides naturally into three core topics:

1. Concepts and techniques.
2. Algorithms and data structures.
3. Program design and software engineering.

The book gives a thorough treatment of topic (1) and an introduction to (2) and (3). In which order should the topics be given? There is a strong interdependency between (1) and (3). Experience shows that program design should be taught early on, so that students avoid bad habits. However, this is only part of the story since students need to know about concepts to express their designs. Parnas has used an approach that starts with topic (3) and uses an imperative computation model [143]. Because this book uses many computation models, we recommend using it to teach (1) and (3) concurrently, introducing new concepts and design principles gradually. In the informatics program at UCL, we attribute eight semester-hours to each topic. This includes lectures and lab sessions. Together the three topics comprise one sixth of the full informatics curriculum for licentiate and engineering degrees.

There is another point we would like to make, which concerns how to teach concurrent programming. In a traditional informatics curriculum, concurrency is taught by extending a stateful model, just as Chapter 8 extends Chapter 6. This is rightly considered to be complex and difficult to program with. There are other, simpler forms of concurrent programming. The declarative concurrency of Chapter 4 is much simpler to program with and can often be used in place of stateful concurrency (see the quote that starts Chapter 4). Stream concurrency, a simple form of declarative concurrency, has been taught in first-year courses at MIT and other institutions. Another simple form of concurrency, message passing between threads, is explained in Chapter 5. We suggest that both declarative concurrency and message-passing concurrency be part of the standard curriculum and be taught before stateful concurrency.

Courses

We have used the book as a textbook for several courses ranging from second-year undergraduate to graduate courses [200, 199, 157]. In its present form, this book is *not* intended as a first programming course, but the approach could likely be adapted for such a course.[2] Students should have a small amount of previous programming experience (e.g., a practical introduction to programming and knowledge of simple data structures such as sequences, sets, stacks, trees, and graphs) and a small amount of mathematical maturity (e.g., a first course on analysis, discrete mathematics, or algebra). The book has enough material for at least four semester-hours worth of lectures and as many lab sessions. Some of the possible courses are:

- An undergraduate course on programming concepts and techniques. Chapter 1 gives a light introduction. The course continues with Chapters 2–8. Depending on the desired depth of coverage, more or less emphasis can be put on algorithms (to teach algorithms along with programming), concurrency (which can be left out completely, if so desired), or formal semantics (to make intuitions precise).

- An undergraduate course on applied programming models. This includes relational programming (Chapter 9), specific programming languages (especially Erlang, Haskell, Java, and Prolog), graphical user interface programming (Chapter 10), distributed programming (Chapter 11), and constraint programming (Chapter 12). This course is a natural sequel to the previous one.

- An undergraduate course on concurrent and distributed programming (Chapters 4, 5, 8, and 11). Students should have some programming experience. The course can start with small parts of Chapters 2, 3, 6, and 7 to introduce declarative and stateful programming.

- A graduate course on computation models (the whole book, including the semantics in Chapter 13). The course can concentrate on the relationships between the models and on their semantics.

The book's Web site has more information on courses including transparencies and lab assignments for some of them. The Web site has an animated interpreter done by Christian Schulte that shows how the kernel languages execute according to the abstract machine semantics. The book can be used as a complement to other courses:

- Part of an undergraduate course on constraint programming (Chapters 4, 9, and 12).

- Part of a graduate course on intelligent collaborative applications (parts of the whole book, with emphasis on Part III). If desired, the book can be complemented by texts on artificial intelligence (e.g., [160]) or multi-agent systems (e.g., [205]).

[2]We will gladly help anyone willing to tackle this adaptation.

- Part of an undergraduate course on semantics. All the models are formally defined in the chapters that introduce them, and this semantics is sharpened in Chapter 13. This gives a real-sized case study of how to define the semantics of a complete modern programming language.

The book, while it has a solid theoretical underpinning, is intended to give a *practical* education in these subjects. Each chapter has many program fragments, all of which can be executed on the Mozart system (see below). With these fragments, course lectures can have live interactive demonstrations of the concepts. We find that students very much appreciate this style of lecture.

Each chapter ends with a set of exercises that usually involve some programming. They can be solved on the Mozart system. To best learn the material in the chapter, we encourage students to do as many exercises as possible. Exercises marked *(advanced exercise)* can take from several days up to several weeks. Exercises marked *(research project)* are open ended and can result in significant research contributions.

Software

A useful feature of the book is that all program fragments can be run on a software platform, the *Mozart Programming System*. Mozart is a full-featured production-quality programming system that comes with an interactive incremental development environment and a full set of tools. It compiles to an efficient platform-independent bytecode that runs on many varieties of Unix and Windows, and on Mac OS X. Distributed programs can be spread out over all these systems. The Mozart Web site, `http://www.mozart-oz.org`, has complete information including downloadable binaries, documentation, scientific publications, source code, and mailing lists.

The Mozart system efficiently implements all the computation models covered in the book. This makes it ideal for using models together in the same program and for comparing models by writing programs to solve a problem in different models. Because each model is implemented efficiently, whole programs can be written in just one model. Other models can be brought in later, if needed, in a pedagogically justified way. For example, programs can be completely written in an object-oriented style, complemented by small declarative components where they are most useful.

The Mozart system is the result of a long-term development effort by the Mozart Consortium, an informal research and development collaboration of three laboratories. It has been under continuing development since 1991. The system is released with full source code under an Open Source license agreement. The first public release was in 1995. The first public release with distribution support was in 1999. The book is based on an ideal implementation that is close to Mozart version 1.3.0, released in 2003. The differences between the ideal implementation and Mozart are listed on the book's Web site.

History and acknowledgements

The ideas in this book did not come easily. They came after more than a decade of discussion, programming, evaluation, throwing out the bad, and bringing in the good and convincing others that it is good. Many people contributed ideas, implementations, tools, and applications. We are lucky to have had a coherent vision among our colleagues for such a long period. Thanks to this, we have been able to make progress.

Our main research vehicle and "testbed" of new ideas is the Mozart system, which implements the Oz language. The system's main designers and developers are and were (in alphabetic order): Per Brand, Thorsten Brunklaus, Denys Duchier, Donatien Grolaux, Seif Haridi, Dragan Havelka, Martin Henz, Erik Klintskog, Leif Kornstaedt, Michael Mehl, Martin Müller, Tobias Müller, Anna Neiderud, Konstantin Popov, Ralf Scheidhauer, Christian Schulte, Gert Smolka, Peter Van Roy, and Jörg Würtz. Other important contributors are and were (in alphabetic order): Iliès Alouini, Thorsten Brunklaus, Raphaël Collet, Frej Drejhammer, Sameh El-Ansary, Nils Franzén, Kevin Glynn, Martin Homik, Simon Lindblom, Benjamin Lorenz, Valentin Mesaros, and Andreas Simon.

We would also like to thank the following researchers and indirect contributors: Hassan Aït-Kaci, Joe Armstrong, Joachim Durchholz, Andreas Franke, Claire Gardent, Fredrik Holmgren, Sverker Janson, Torbjörn Lager, Elie Milgrom, Johan Montelius, Al-Metwally Mostafa, Joachim Niehren, Luc Onana, Marc-Antoine Parent, Dave Parnas, Mathias Picker, Andreas Podelski, Christophe Ponsard, Mahmoud Rafea, Juris Reinfelds, Thomas Sjöland, Fred Spiessens, Joe Turner, and Jean Vanderdonckt.

We give a special thanks to the following people for their help with material related to the book. We thank Raphaël Collet for co-authoring Chapters 12 and 13 and for his work on the practical part of LINF1251, a course taught at UCL. We thank Donatien Grolaux for three GUI case studies (used in Sections 10.3.2–10.3.4). We thank Kevin Glynn for writing the Haskell introduction (Section 4.8). We thank Frej Drejhammar, Sameh El-Ansary, and Dragan Havelka for their work on the practical part of DatalogiII, a course taught at KTH. We thank Christian Schulte who was responsible for completely rethinking and redeveloping a subsequent edition of DatalogiII and for his comments on a draft of the book. We thank Ali Ghodsi, Johan Montelius, and the other three assistants for their work on the practical part of this edition. We thank Luis Quesada and Kevin Glynn for their work on the practical part of INGI2131, a course taught at UCL. We thank Bruno Carton, Raphaël Collet, Kevin Glynn, Donatien Grolaux, Stefano Gualandi, Valentin Mesaros, Al-Metwally Mostafa, Luis Quesada, and Fred Spiessens for their efforts in proofreading and testing the example programs. Finally, we thank the members of the Department of Computing Science and Engineering at UCL, the Swedish Institute of Computer Science, and the Department of Microelectronics and Information Technology at KTH. We apologize to anyone we may have inadvertently omitted.

How did we manage to keep the result so simple with such a large crowd of developers working together? No miracle, but the consequence of a strong vision and a carefully crafted design methodology that took more than a decade to create and polish (see [196] for a summary; we can summarize it as "a design is either simple or wrong"). Around 1990, some of us came together with already strong systems building and theoretical backgrounds. These people initiated the ACCLAIM project, funded by the European Union (1991–1994). For some reason, this project became a focal point. Three important milestones among many were the papers by Sverker Janson & Seif Haridi in 1991 [93] (multiple paradigms in AKL), by Gert Smolka in 1995 [180] (building abstractions in Oz), and by Seif Haridi *et al* in 1998 [72] (dependable open distribution in Oz). The first paper on Oz was published in 1993 and already had many important ideas [80]. After ACCLAIM, two laboratories continued working together on the Oz ideas: the Programming Systems Lab (DFKI, Universität des Saarlandes, and Collaborative Research Center SFB 378) in Saarbrücken, Germany, and the Intelligent Systems Laboratory (Swedish Institute of Computer Science), in Stockholm, Sweden.

The Oz language was originally designed by Gert Smolka and his students in the Programming Systems Lab [79, 173, 179, 81, 180, 74, 172]. The well-factorized design of the language and the high quality of its implementation are due in large part to Smolka's inspired leadership and his lab's system-building expertise. Among the developers, we mention Christian Schulte for his role in coordinating general development, Denys Duchier for his active support of users, and Per Brand for his role in coordinating development of the distributed implementation. In 1996, the German and Swedish labs were joined by the Department of Computing Science and Engineering (Université catholique de Louvain), in Louvain-la-Neuve, Belgium, when the first author moved there. Together the three laboratories formed the Mozart Consortium with its neutral Web site `http://www.mozart-oz.org` so that the work would not be tied down to a single institution.

This book was written using LaTeX 2_ε, flex, xfig, xv, vi/vim, emacs, and Mozart, first on a Dell Latitude with Red Hat Linux and KDE, and then on an Apple Macintosh PowerBook G4 with Mac OS X and X11. The first author thanks the Walloon Region of Belgium for their generous support of the Oz/Mozart work at UCL in the PIRATES project.

What's missing

There are two main topics missing from the book:

- *Static typing.* The formalism used in this book is dynamically typed. Despite the advantages of static typing for program verification, security, and implementation efficiency, we barely mention it. The main reason is that the book focuses on expressing computations with programming concepts, with as few restrictions as possible. There is already plenty to say even within this limited scope, as witness the size of the book.

- *Specialized programming techniques.* The set of programming techniques is too vast to explain in one book. In addition to the general techniques explained in this book, each problem domain has its own particular techniques. This book does not cover all of them; attempting to do so would double or triple its size. To make up for this lack, we point the reader to some good books that treat particular problem domains: artificial intelligence techniques [160, 136], algorithms [41], object-oriented design patterns [58], multi-agent programming [205], databases [42], and numerical techniques [153].

Final comments

We have tried to make this book useful both as a textbook and as a reference. It is up to you to judge how well it succeeds in this. Because of its size, it is likely that some errors remain. If you find any, we would appreciate hearing from you. Please send them and all other constructive comments you may have to the following address:

Concepts, Techniques, and Models of Computer Programming
Department of Computing Science and Engineering
Université catholique de Louvain
B-1348 Louvain-la-Neuve, Belgium

As a final word, we would like to thank our families and friends for their support and encouragement during the more than three years it took us to write this book. Seif Haridi would like to give a special thanks to his parents Ali and Amina and to his family Eeva, Rebecca, and Alexander. Peter Van Roy would like to give a special thanks to his parents Frans and Hendrika and to his family Marie-Thérèse, Johan, and Lucile.

Louvain-la-Neuve, Belgium Peter Van Roy
Kista, Sweden Seif Haridi
June 2003