Concepts, Techniques, and Models of Computer Programming

PETER VAN ROY and SEIF HARIDI

Concepts, Techniques, and Models of Computer Programming

by
Peter Van Roy
Seif Haridi

The MIT Press
Cambridge, Massachusetts
London, England

This book was set in LaTeX 2ε by the authors and was printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

Van Roy, Peter.
Concepts, techniques, and models of computer programming / Peter Van Roy, Seif Haridi
p. cm.
Includes bibliographical references and index.
ISBN 0-262-22069-5
1. Computer programming. I. Haridi, Seif. II. Title.

QA76.6.V36 2004
005.1—dc22 2003065140

10 9 8 7 6 5 4 3

Short Contents

	Preface	xiii
	Running the Example Programs	xxix
1	Introduction to Programming Concepts	1
I	GENERAL COMPUTATION MODELS	27
2	Declarative Computation Model	29
3	Declarative Programming Techniques	111
4	Declarative Concurrency	233
5	Message-Passing Concurrency	345
6	Explicit State	405
7	Object-Oriented Programming	489
8	Shared-State Concurrency	569
9	Relational Programming	621
II	SPECIALIZED COMPUTATION MODELS	677
10	Graphical User Interface Programming	679
11	Distributed Programming	707
12	Constraint Programming	749
III	SEMANTICS	777
13	Language Semantics	779

IV APPENDIXES 813

A Mozart System Development Environment 815

B Basic Data Types 819

C Language Syntax 833

D General Computation Model 843

References 853

Index 863

Table of Contents

Preface xiii

Running the Example Programs xxix

1 Introduction to Programming Concepts 1
- 1.1 A calculator . . . 1
- 1.2 Variables . . . 2
- 1.3 Functions . . . 2
- 1.4 Lists . . . 4
- 1.5 Functions over lists . . . 7
- 1.6 Correctness . . . 9
- 1.7 Complexity . . . 10
- 1.8 Lazy evaluation . . . 11
- 1.9 Higher-order programming . . . 13
- 1.10 Concurrency . . . 14
- 1.11 Dataflow . . . 15
- 1.12 Explicit state . . . 16
- 1.13 Objects . . . 17
- 1.14 Classes . . . 18
- 1.15 Nondeterminism and time . . . 20
- 1.16 Atomicity . . . 21
- 1.17 Where do we go from here? . . . 22
- 1.18 Exercises . . . 23

I GENERAL COMPUTATION MODELS 27

2 Declarative Computation Model 29
- 2.1 Defining practical programming languages . . . 30
- 2.2 The single-assignment store . . . 42
- 2.3 Kernel language . . . 49
- 2.4 Kernel language semantics . . . 56
- 2.5 Memory management . . . 72
- 2.6 From kernel language to practical language . . . 79
- 2.7 Exceptions . . . 90

2.8 Advanced topics . . . 96
2.9 Exercises . . . 107

3 Declarative Programming Techniques 111
3.1 What is declarativeness? . . . 114
3.2 Iterative computation . . . 118
3.3 Recursive computation . . . 124
3.4 Programming with recursion . . . 127
3.5 Time and space efficiency . . . 166
3.6 Higher-order programming . . . 177
3.7 Abstract data types . . . 195
3.8 Nondeclarative needs . . . 210
3.9 Program design in the small . . . 218
3.10 Exercises . . . 230

4 Declarative Concurrency 233
4.1 The data-driven concurrent model . . . 235
4.2 Basic thread programming techniques . . . 246
4.3 Streams . . . 256
4.4 Using the declarative concurrent model directly . . . 272
4.5 Lazy execution . . . 278
4.6 Soft real-time programming . . . 304
4.7 The Haskell language . . . 308
4.8 Limitations and extensions of declarative programming . . . 313
4.9 Advanced topics . . . 326
4.10 Historical notes . . . 337
4.11 Exercises . . . 338

5 Message-Passing Concurrency 345
5.1 The message-passing concurrent model . . . 347
5.2 Port objects . . . 350
5.3 Simple message protocols . . . 353
5.4 Program design for concurrency . . . 362
5.5 Lift control system . . . 365
5.6 Using the message-passing model directly . . . 377
5.7 The Erlang language . . . 386
5.8 Advanced topic . . . 394
5.9 Exercises . . . 399

6 Explicit State 405
6.1 What is state? . . . 408
6.2 State and system building . . . 410
6.3 The declarative model with explicit state . . . 413
6.4 Data abstraction . . . 419
6.5 Stateful collections . . . 435

6.6 Reasoning with state . . . 440
6.7 Program design in the large . . . 450
6.8 Case studies . . . 463
6.9 Advanced topics . . . 479
6.10 Exercises . . . 482

7 Object-Oriented Programming 489
7.1 Inheritance . . . 491
7.2 Classes as complete data abstractions . . . 492
7.3 Classes as incremental data abstractions . . . 502
7.4 Programming with inheritance . . . 518
7.5 Relation to other computation models . . . 537
7.6 Implementing the object system . . . 545
7.7 The Java language (sequential part) . . . 551
7.8 Active objects . . . 556
7.9 Exercises . . . 567

8 Shared-State Concurrency 569
8.1 The shared-state concurrent model . . . 573
8.2 Programming with concurrency . . . 573
8.3 Locks . . . 582
8.4 Monitors . . . 592
8.5 Transactions . . . 600
8.6 The Java language (concurrent part) . . . 615
8.7 Exercises . . . 618

9 Relational Programming 621
9.1 The relational computation model . . . 623
9.2 Further examples . . . 627
9.3 Relation to logic programming . . . 631
9.4 Natural language parsing . . . 641
9.5 A grammar interpreter . . . 650
9.6 Databases . . . 654
9.7 The Prolog language . . . 660
9.8 Exercises . . . 671

II SPECIALIZED COMPUTATION MODELS 677

10 Graphical User Interface Programming 679
10.1 The declarative/procedural approach . . . 681
10.2 Using the declarative/procedural approach . . . 682
10.3 The Prototyper interactive learning tool . . . 689
10.4 Case studies . . . 690
10.5 Implementing the GUI tool . . . 703

10.6 Exercises . . . 703

11 Distributed Programming **707**
11.1 Taxonomy of distributed systems . . . 710
11.2 The distribution model . . . 712
11.3 Distribution of declarative data . . . 714
11.4 Distribution of state . . . 720
11.5 Network awareness . . . 723
11.6 Common distributed programming patterns . . . 724
11.7 Distribution protocols . . . 732
11.8 Partial failure . . . 739
11.9 Security . . . 743
11.10 Building applications . . . 745
11.11 Exercises . . . 746

12 Constraint Programming **749**
12.1 Propagate-and-search . . . 750
12.2 Programming techniques . . . 755
12.3 The constraint-based computation model . . . 758
12.4 Defining and using computation spaces . . . 762
12.5 Implementing the relational computation model . . . 772
12.6 Exercises . . . 774

III SEMANTICS **777**

13 Language Semantics **779**
13.1 The general computation model . . . 780
13.2 Declarative concurrency . . . 804
13.3 Eight computation models . . . 806
13.4 Semantics of common abstractions . . . 808
13.5 Historical notes . . . 808
13.6 Exercises . . . 809

IV APPENDIXES **813**

A Mozart System Development Environment **815**
A.1 Interactive interface . . . 815
A.2 Command line interface . . . 817

B Basic Data Types **819**
B.1 Numbers (integers, floats, and characters) . . . 819
B.2 Literals (atoms and names) . . . 824
B.3 Records and tuples . . . 825

B.4 Chunks (limited records) 828
B.5 Lists 828
B.6 Strings 830
B.7 Virtual strings 831

C Language Syntax **833**
C.1 Interactive statements 834
C.2 Statements and expressions 834
C.3 Nonterminals for statements and expressions 836
C.4 Operators 836
C.5 Keywords 839
C.6 Lexical syntax 839

D General Computation Model **843**
D.1 Creative extension principle 844
D.2 Kernel language 845
D.3 Concepts 846
D.4 Different forms of state 849
D.5 Other concepts 850
D.6 Layered language design 850

References **853**

Index **863**

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*, Seymour Papert (1980)

One approach to the study of 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 in using 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 be directly useful to the programmer. We 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 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 criterion 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, adding just 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 should we add a concept to a model and which concept should we add? We touch on these questions many times. The main criterion 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 has been 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 judgment.

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 for 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 is 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 into 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 language(s) (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 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 (e.g., 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 in the 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 [70].

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 [43], and Leda has elements that are functional, object-oriented, and logical [31]. 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 [153]. 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 [63], [70], and [80] can be an inspiration to programmers.

Main features

Pedagogical approach

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

- The computation-based approach presents programming as a way to define

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

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 [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 [148]. Programs are constructed with a building-block approach, using lower-level abstractions to build higher-level 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 the 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 or 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 and imperative models (and between concurrent and 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 the 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 [43] and Standard ML [145, 213]) and more recently with concurrency (e.g., Concurrent ML [176] and Concurrent Haskell [167, 165]).

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 [66]. 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 4.8). 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 flip side is that it is not clear by inspection whether a 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 [217]).

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 judgment 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 [161]. Because this book uses many computation models, we recommend using it to teach (1) and (3) concurrently, introducing new concepts and design principles together. In the informatics program at the Université catholique de Louvain at Louvain-la-Neuve, Belgium (UCL), we attribute eight semester-hours to each topic. This includes lectures and lab sessions. Together the three topics make up 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 epigraph 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 [175, 220, 221]. In its present form, the book is not intended as a first programming course, but the approach could likely be adapted for such a course.[2] Students should have some basic programming

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

experience (e.g., a practical introduction to programming and knowledge of simple data structures such as sequences, sets, and stacks) and some basic mathematical knowledge (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 through 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 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 II). If desired, the book can be complemented by texts on artificial intelligence (e.g., [179]) or multi-agent systems (e.g., [226]).
- 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 implements efficiently 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 April 2004. The differences between the ideal implementation and Mozart are listed on the book's Web site.

History and acknowledgments

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 "test bed" of new ideas is the Mozart system, which implements the Oz language. The system's main designers and developers are (in alphabetical order) Per Brand, Thorsten Brunklaus, Denys Duchier, Kevin Glynn, 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 (in alphabetical order) Iliès Alouini, Raphaël Collet, Frej Drejhammar, Sameh El-Ansary, Nils Franzén, Martin Homik, Simon Lindblom, Benjamin Lorenz, Valentin Mesaros, and Andreas Simon. We thank Konstantin Popov and Kevin Glynn for managing the release of Mozart version 1.3.0, which is designed to accompany the book.

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 special thanks to the following people for their help with material related to the book. Raphaël Collet for co-authoring chapters 12 and 13, for his work on the practical part of LINF1251, a course taught at UCL, and for his help with the LaTeX 2_ε formatting. Donatien Grolaux for three graphical user interface case studies (used in sections 10.4.2–10.4.4). Kevin Glynn for writing the Haskell introduction (section 4.7). William Cook for his comments on data abstraction. Frej Drejhammar, Sameh El-Ansary, and Dragan Havelka for their help with the practical part of Datalogi II, a course taught at KTH (the Royal Institute of Technology, Stockholm). Christian Schulte for completely rethinking and redeveloping a subsequent edition of Datalogi II and for his comments on a draft of the book. Ali Ghodsi, Johan Montelius, and the other three assistants for their help with the practical part of this edition. Luis Quesada and Kevin Glynn for their work on the practical part of INGI2131, a course taught at UCL. 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. We thank other people too numerous to mention for their comments on the book. Finally, we thank the members of the Department of Computing Science and Engineering at UCL, SICS (the Swedish Institute of Computer Science, Stockholm), 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.[3] Around 1990, some of us came together with already strong system-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 and Seif Haridi in 1991 [105] (multiple paradigms in the Andorra Kernel Language AKL), by Gert Smolka in 1995 [199] (building abstractions in Oz), and by Seif Haridi et al. in 1998 [83] (dependable open distribution in Oz). The first paper on Oz was published in 1993 and already had many important ideas [89]. After ACCLAIM, two laboratories continued working together on the Oz ideas: the Programming Systems Lab (DFKI, Saarland University, and Collaborative Research Center SFB 378) at Saarbrücken, Germany, and the Intelligent Systems Laboratory at SICS.

The Oz language was originally designed by Gert Smolka and his students in the Programming Systems Lab [85, 89, 90, 190, 192, 198, 199]. 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 at UCL 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$_\varepsilon$, 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 screenshots were taken on a Sun workstation running Solaris. The first author thanks the Walloon Region of Belgium for their generous support of the Oz/Mozart work at UCL in the PIRATES and MILOS projects.

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:

3. We can summarize the methodology in two rules (see [217] for more information). First, a new abstraction must either simplify the system or greatly increase its expressive power. Second, a new abstraction must have both an efficient implementation and a simple formalization.

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 four 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
September 2003

Running the Example Programs

This book gives many example programs and program fragments. All of these can be run on the Mozart Programming System. To make this as easy as possible, please keep the following points in mind:

- The Mozart system can be downloaded without charge from the Mozart Consortium Web site `http://www.mozart-oz.org`. Releases exist for various flavors of Windows and Unix and for Mac OS X.
- All examples, except those intended for standalone applications, can be run in Mozart's interactive development environment. Appendix A gives an introduction to this environment.
- New variables in the interactive examples must be declared with the **`declare`** statement. The examples of chapter 1 show how to do this. Forgetting these declarations can result in strange errors if older versions of the variables exist. Starting with chapter 2 and for all following chapters, the **`declare`** statement is omitted in the text when it is obvious what the new variables are. It should be added to run the examples.
- Some chapters use operations that are not part of the standard Mozart release. The source code for these additional operations (along with much other useful material) is given on the book's Web site. We recommend putting these definitions in your `.ozrc` file, so they will be loaded automatically when the system starts up.
- The book occasionally gives screenshots and timing measurements of programs. The screenshots were taken on a Sun workstation running Solaris and the timing measurements were done on Mozart 1.1.0 under Red Hat Linux release 6.1 on a Dell Latitude CPx notebook computer with Pentium III processor at 500 MHz, unless otherwise indicated. Screenshot appearance and timing numbers may vary on your system.
- The book assumes an ideal implementation whose semantics is given in chapter 13 (general computation model) and chapter 12 (computation spaces). There are a few differences between this ideal implementation and the Mozart system. They are explained on the book's Web site.

1 Introduction to Programming Concepts

There is no royal road to geometry.
— Euclid's reply to Ptolemy, Euclid (*fl. c.* 300 B.C.)

Just follow the yellow brick road.
— *The Wonderful Wizard of Oz*, L. Frank Baum (1856–1919)

Programming is telling a computer how it should do its job. This chapter gives a gentle, hands-on introduction to many of the most important concepts in programming. We assume you have had some previous exposure to computers. We use the interactive interface of Mozart to introduce programming concepts in a progressive way. We encourage you to try the examples in this chapter on a running Mozart system.

This introduction only scratches the surface of the programming concepts we will see in this book. Later chapters give a deep understanding of these concepts and add many other concepts and techniques.

1.1 A calculator

Let us start by using the system to do calculations. Start the Mozart system by typing:

```
oz
```

or by double-clicking a Mozart icon. This opens an editor window with two frames. In the top frame, type the following line:

```
{Browse 9999*9999}
```

Use the mouse to select this line. Now go to the **Oz** menu and select **Feed Region**. This feeds the selected text to the system. The system then does the calculation `9999*9999` and displays the result, `99980001`, in a special window called the browser. The curly braces { ... } are used for a procedure or function call. `Browse` is a procedure with one argument, which is called as `{Browse X}`. This opens the browser window, if it is not already open, and displays `X` in it.

1.2 Variables

While working with the calculator, we would like to remember an old result, so that we can use it later without retyping it. We can do this by declaring a variable:

```
declare
V=9999*9999
```

This declares `V` and binds it to `99980001`. We can use this variable later on:

```
{Browse V*V}
```

This displays the answer `9996000599960001`. Variables are just shortcuts for values. They cannot be assigned more than once. But you can declare another variable with the same name as a previous one. The previous variable then becomes inaccessible. Previous calculations that used it are not changed. This is because there are two concepts hiding behind the word "variable":

- The identifier. This is what you type in. Variables start with a capital letter and can be followed by any number of letters or digits. For example, the character sequence `Var1` can be a variable identifier.
- The store variable. This is what the system uses to calculate with. It is part of the system's memory, which we call its store.

The **declare** statement creates a new store variable and makes the variable identifier refer to it. Previous calculations using the same identifier are not changed because the identifier refers to another store variable.

1.3 Functions

Let us do a more involved calculation. Assume we want to calculate the factorial function $n!$, which is defined as $1 \times 2 \times \cdots \times (n-1) \times n$. This gives the number of permutations of n items, i.e., the number of different ways these items can be put in a row. Factorial of 10 is:

```
{Browse 1*2*3*4*5*6*7*8*9*10}
```

This displays `3628800`. What if we want to calculate the factorial of 100? We would like the system to do the tedious work of typing in all the integers from 1 to 100. We will do more: we will tell the system how to calculate the factorial of any n. We do this by defining a function:

```
declare
fun {Fact N}
   if N==0 then 1 else N*{Fact N-1} end
end
```

The **declare** statement creates the new variable `Fact`. The **fun** statement defines a function. The variable `Fact` is bound to the function. The function has one

argument `N`, which is a local variable, i.e., it is known only inside the function body. Each time we call the function a new local variable is created.

Recursion

The function body is an instruction called an **`if`** expression. When the function is called, then the **`if`** expression does the following steps:

- It first checks whether `N` is equal to 0 by doing the test `N==0`.
- If the test succeeds, then the expression after the **`then`** is calculated. This just returns the number 1. This is because the factorial of 0 is 1.
- If the test fails, then the expression after the **`else`** is calculated. That is, if `N` is not 0, then the expression `N*{Fact N-1}` is calculated. This expression uses `Fact`, the very function we are defining! This is called recursion. It is perfectly normal and no cause for alarm.

`Fact` uses the following mathematical definition of factorial:

$0! = 1$

$n! = n \times (n-1)!$ if $n > 0$

This definition is recursive because the factorial of `N` is `N` times the factorial of `N-1`. Let us try out the function `Fact`:

```
{Browse {Fact 10}}
```

This should display `3628800` as before. This gives us confidence that `Fact` is doing the right calculation. Let us try a bigger input:

```
{Browse {Fact 100}}
```

This will display a huge number (which we show in groups of five digits to improve readability):

```
                                                  933 26215
44394 41526 81699 23885 62667 00490 71596 82643 81621 46859
29638 95217 59999 32299 15608 94146 39761 56518 28625 36979
20827 22375 82511 85210 91686 40000 00000 00000 00000 00000
```

This is an example of arbitrary precision arithmetic, sometimes called "infinite precision," although it is not infinite. The precision is limited by how much memory your system has. A typical low-cost personal computer with 256 MB of memory can handle hundreds of thousands of digits. The skeptical reader will ask: is this huge number really the factorial of 100? How can we tell? Doing the calculation by hand would take a long time and probably be incorrect. We will see later on how to gain confidence that the system is doing the right thing.

Combinations

Let us write a function to calculate the number of combinations of k items taken from n. This is equal to the number of subsets of size k that can be made from a set of size n. This is written $\binom{n}{k}$ in mathematical notation and pronounced "n choose k." It can be defined as follows using the factorial:

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$

which leads naturally to the following function:

```
declare
fun {Comb N K}
   {Fact N} div ({Fact K}*{Fact N-K})
end
```

For example, `{Comb 10 3}` is 120, which is the number of ways that 3 items can be taken from 10. This is not the most efficient way to write `Comb`, but it is probably the simplest.

Functional abstraction

The definition of `Comb` uses the existing function `Fact` in its definition. It is always possible to use existing functions when defining new functions. Using functions to build abstractions is called functional abstraction. In this way, programs are like onions, with layers upon layers of functions calling functions. This style of programming is covered in chapter 3.

1.4 Lists

Now we can calculate functions of integers. But an integer is really not very much to look at. Say we want to calculate with lots of integers. For example, we would like to calculate Pascal's triangle[1]:

```
                1
              1   1
            1   2   1
          1   3   3   1
        1   4   6   4   1
. . . . . . . . . . . . .
```

1. Pascal's triangle is a key concept in combinatorics. The elements of the nth row are the combinations $\binom{n}{k}$, where k ranges from 0 to n. This is closely related to the binomial theorem, which states $(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^k y^{(n-k)}$ for integer $n \geq 0$.

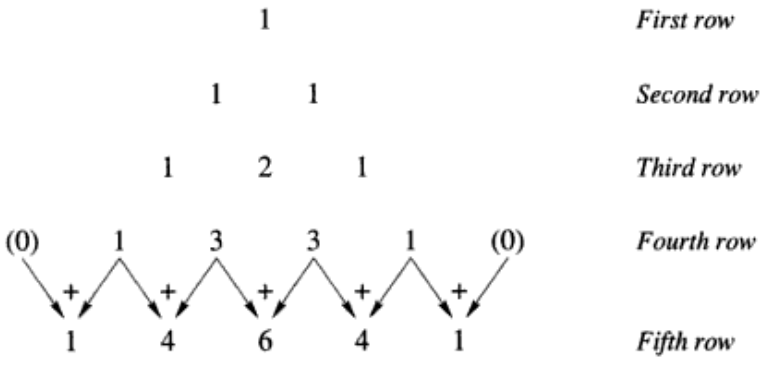


Figure 1.2: Calculating the fifth row of Pascal's triangle.

```
declare
H=5
T=[6 7 8]
{Browse H|T}
```

The list `H|T` can be written `[5 6 7 8]`. It has head `5` and tail `[6 7 8]`. The cons `H|T` can be taken apart, to get back the head and tail:

```
declare
L=[5 6 7 8]
{Browse L.1}
{Browse L.2}
```

This uses the dot operator ".", which is used to select the first or second argument of a list pair. Doing `L.1` gives the head of `L`, the integer `5`. Doing `L.2` gives the tail of `L`, the list `[6 7 8]`. Figure 1.1 gives a picture: `L` is a chain in which each link has one list element and `nil` marks the end. Doing `L.1` gets the first element and doing `L.2` gets the rest of the chain.

Pattern matching

A more compact way to take apart a list is by using the **`case`** instruction, which gets both head and tail in one step:

```
declare
L=[5 6 7 8]
case L of H|T then {Browse H} {Browse T} end
```

This displays `5` and `[6 7 8]`, just like before. The **`case`** instruction declares two local variables, `H` and `T`, and binds them to the head and tail of the list `L`. We say the **`case`** instruction does pattern matching, because it decomposes `L` according to the "pattern" `H|T`. Local variables declared with a **`case`** are just like variables declared with **`declare`**, except that the variable exists only in the body of the **`case`** statement, i.e., between the **`then`** and the **`end`**.

```
fun {ShiftLeft L}
   case L of H|T then
      H|{ShiftLeft T}
   else [0] end
end

fun {ShiftRight L} 0|L end
```

`ShiftRight` just adds a zero to the left. `ShiftLeft` traverses `L` one element at a time and builds the output one element at a time. We have added an **`else`** to the **`case`** instruction. This is similar to an **`else`** in an **`if`**: it is executed if the pattern of the **`case`** does not match. That is, when `L` is empty, then the output is `[0]`, i.e., a list with just zero inside.

Here is `AddList`:

```
fun {AddList L1 L2}
   case L1 of H1|T1 then
      case L2 of H2|T2 then
         H1+H2|{AddList T1 T2}
      end
   else nil end
end
```

This is the most complicated function we have seen so far. It uses two **`case`** instructions, one inside another, because we have to take apart two lists, `L1` and `L2`. Now we have the complete definition of `Pascal`. We can calculate any row of Pascal's triangle. For example, calling `{Pascal 20}` returns the 20th row:

```
[1 19 171 969 3876 11628 27132 50388 75582 92378
 92378 75582 50388 27132 11628 3876 969 171 19 1]
```

Is this answer correct? How can we tell? It looks right: it is symmetric (reversing the list gives the same list) and the first and second arguments are 1 and 19, which are right. Looking at figure 1.2, it is easy to see that the second element of the nth row is always $n - 1$ (it is always one more than the previous row and it starts out zero for the first row). In the next section, we will see how to reason about correctness.

Top-down software development

Let us summarize the methodology we used to write `Pascal`:

- The first step is to understand how to do the calculation by hand.
- The second step is to write a main function to solve the problem, assuming that some auxiliary functions are known (here, `ShiftLeft`, `ShiftRight`, and `AddList`).
- The third step is to complete the solution by writing the auxiliary functions.

The methodology of first writing the main function and filling in the blanks afterward is known as *top-down* software development. It is one of the best known approaches to program design, but it gives only part of the story as we shall see.

1.6 Correctness

A program is correct if it does what we would like it to do. How can we tell whether a program is correct? Usually it is impossible to duplicate the program's calculation by hand. We need other ways. One simple way, which we used before, is to verify that the program is correct for outputs that we know. This increases confidence in the program. But it does not go very far. To prove correctness in general, we have to reason about the program. This means three things:

- We need a mathematical model of the operations of the programming language, defining what they should do. This model is called the language's semantics.
- We need to define what we would like the program to do. Usually, this is a mathematical definition of the inputs that the program needs and the output that it calculates. This is called the program's specification.
- We use mathematical techniques to reason about the program, using the semantics. We would like to demonstrate that the program satisfies the specification.

A program that is proved correct can still give incorrect results, if the system on which it runs is incorrectly implemented. How can we be confident that the system satisfies the semantics? Verifying this is a major undertaking: it means verifying the compiler, the run-time system, the operating system, the hardware, and the physics upon which the hardware is based! These are all important tasks, but they are beyond the scope of the book. We place our trust in the Mozart developers, software companies, hardware manufacturers, and physicists.[3]

Mathematical induction

One very useful technique is mathematical induction. This proceeds in two steps. We first show that the program is correct for the simplest case. Then we show that, if the program is correct for a given case, then it is correct for the next case. If we can be sure that all cases are eventually covered, then mathematical induction lets us conclude that the program is always correct. This technique can be applied for integers and lists:

- For integers, the simplest case is 0 and for a given integer n the next case is $n+1$.
- For lists, the simplest case is `nil` (the empty list) and for a given list `T` the next case is `H|T` (with no conditions on `H`).

Let us see how induction works for the factorial function:

- `{Fact 0}` returns the correct answer, namely 1.

3. Some would say that this is foolish. Paraphrasing Thomas Jefferson, they would say that the price of correctness is eternal vigilance.

- Assume that {Fact N-1} is correct. Then look at the call {Fact N}. We see that the **if** instruction takes the **else** case (since N is not zero), and calculates N*{Fact N-1}. By hypothesis, {Fact N-1} returns the right answer. Therefore, assuming that the multiplication is correct, {Fact N} also returns the right answer.

This reasoning uses the mathematical definition of factorial, namely $n! = n \times (n-1)!$ if $n > 0$, and $0! = 1$. Later in the book we will see more sophisticated reasoning techniques. But the basic approach is always the same: start with the language semantics and problem specification, and use mathematical reasoning to show that the program correctly implements the specification.

1.7 Complexity

The Pascal function we defined above gets very slow if we try to calculate higher-numbered rows. Row 20 takes a second or two. Row 30 takes many minutes.[4] If you try it, wait patiently for the result. How come it takes this much time? Let us look again at the function Pascal:

```
fun {Pascal N}
   if N==1 then [1]
   else
     {AddList {ShiftLeft {Pascal N-1}} {ShiftRight {Pascal N-1}}}
   end
end
```

Calling {Pascal N} will call {Pascal N-1} two times. Therefore, calling {Pascal 30} will call {Pascal 29} twice, giving four calls to {Pascal 28}, eight to {Pascal 27}, and so forth, doubling with each lower row. This gives 2^{29} calls to {Pascal 1}, which is about half a billion. No wonder that {Pascal 30} is slow. Can we speed it up? Yes, there is an easy way: just call {Pascal N-1} once instead of twice. The second call gives the same result as the first. If we could just remember it, then one call would be enough. We can remember it by using a local variable. Here is a new function, FastPascal, that uses a local variable:

```
fun {FastPascal N}
   if N==1 then [1]
   else L in
      L={FastPascal N-1}
      {AddList {ShiftLeft L} {ShiftRight L}}
   end
end
```

We declare the local variable L by adding "L **in**" to the **else** part. This is just like using **declare**, except that the identifier can only be used between the **else** and the **end**. We bind L to the result of {FastPascal N-1}. Now we can use L wherever we need it. How fast is FastPascal? Try calculating row 30. This takes minutes

4. These times may vary depending on the speed of your machine.

with `Pascal`, but is done practically instantaneously with `FastPascal`. A lesson we can learn from this example is that using a good algorithm is more important than having the best possible compiler or fastest machine.

Run-time guarantees of execution time

As this example shows, it is important to know something about a program's execution time. Knowing the exact time is less important than knowing that the time will not blow up with input size. The execution time of a program as a function of input size, up to a constant factor, is called the program's *time complexity*. What this function is depends on how the input size is measured. We assume that it is measured in a way that makes sense for how the program is used. For example, we take the input size of `{Pascal N}` to be simply the integer `N` (and not, e.g., the amount of memory needed to store `N`).

The time complexity of `{Pascal N}` is proportional to 2^n. This is an exponential function in n, which grows very quickly as n increases. What is the time complexity of `{FastPascal N}`? There are n recursive calls and each call takes time proportional to n. The time complexity is therefore proportional to n^2. This is a polynomial function in n, which grows at a much slower rate than an exponential function. Programs whose time complexity is exponential are impractical except for very small inputs. Programs whose time complexity is a low-order polynomial are practical.

1.8 Lazy evaluation

The functions we have written so far will do their calculation as soon as they are called. This is called eager evaluation. There is another way to evaluate functions called lazy evaluation.[5] In lazy evaluation, a calculation is done only when the result is needed. This is covered in chapter 4 (see section 4.5). Here is a simple lazy function that calculates a list of integers:

```
fun lazy {Ints N}
   N|{Ints N+1}
end
```

Calling `{Ints 0}` calculates the infinite list `0|1|2|3|4|5|...`. This looks like an infinite loop, but it is not. The `lazy` annotation ensures that the function will only be evaluated when it is needed. This is one of the advantages of lazy evaluation: we can calculate with potentially infinite data structures without any loop boundary conditions. For example:

5. Eager and lazy evaluation are sometimes called data-driven and demand-driven evaluation, respectively.

```
fun {PascalList2 N Row}
   if N==1 then [Row]
   else
      Row|{PascalList2 N-1
             {AddList {ShiftLeft Row} {ShiftRight Row}}}
   end
end
```

We can display 10 rows by calling `{Browse {PascalList2 10 [1]}}`. But what if later on we decide that we need 11 rows? We would have to call `PascalList2` again, with argument 11. This would redo all the work of defining the first 10 rows. The lazy version avoids redoing all this work. It is always ready to continue where it left off.

1.9 Higher-order programming

We have written an efficient function, `FastPascal`, that calculates rows of Pascal's triangle. Now we would like to experiment with variations on Pascal's triangle. For example, instead of adding numbers to get each row, we would like to subtract them, exclusive-or them (to calculate just whether they are odd or even), or many other possibilities. One way to do this is to write a new version of `FastPascal` for each variation. But this quickly becomes tiresome. Is it possible to have just a single version that can be used for all variations? This is indeed possible. Let us call it `GenericPascal`. Whenever we call it, we pass it the customizing function (adding, exclusive-oring, etc.) as an argument. The ability to pass functions as arguments is known as higher-order programming.

Here is the definition of `GenericPascal`. It has one extra argument `Op` to hold the function that calculates each number:

```
fun {GenericPascal Op N}
   if N==1 then [1]
   else L in
      L={GenericPascal Op N-1}
      {OpList Op {ShiftLeft L} {ShiftRight L}}
   end
end
```

`AddList` is replaced by `OpList`. The extra argument `Op` is passed to `OpList`. `ShiftLeft` and `ShiftRight` do not need to know `Op`, so we can use the old versions. Here is the definition of `OpList`:

```
fun {OpList Op L1 L2}
   case L1 of H1|T1 then
      case L2 of H2|T2 then
         {Op H1 H2}|{OpList Op T1 T2}
      end
   else nil end
end
```

Instead of doing the addition `H1+H2`, this version does `{Op H1 H2}`.

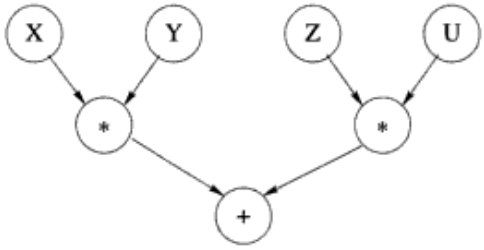


Figure 1.3: A simple example of dataflow execution.

among the activities, unless the programmer decides that they need to communicate. This is how the real world works outside of the system. We would like to be able to do this inside the system as well.

We introduce concurrency by creating threads. A thread is simply an executing program like the functions we saw before. The difference is that a program can have more than one thread. Threads are created with the **thread** instruction. Do you remember how slow the original `Pascal` function was? We can call `Pascal` inside its own thread. This means that it will not keep other calculations from continuing. They may slow down, if `Pascal` really has a lot of work to do. This is because the threads share the same underlying computer. But none of the threads will stop. Here is an example:

```
thread P in
   P={Pascal 30}
   {Browse P}
end
{Browse 99*99}
```

This creates a new thread. Inside this new thread, we call `{Pascal 30}` and then call `Browse` to display the result. The new thread has a lot of work to do. But this does not keep the system from displaying `99*99` immediately.

1.11 Dataflow

What happens if an operation tries to use a variable that is not yet bound? From a purely aesthetic point of view, it would be nice if the operation would simply wait. Perhaps some other thread will bind the variable, and then the operation can continue. This civilized behavior is known as dataflow. Figure 1.3 gives a simple example: the two multiplications wait until their arguments are bound and the addition waits until the multiplications complete. As we will see later in the book, there are many good reasons to have dataflow behavior. For now, let us see how dataflow and concurrency work together. Take, e.g.:

```
declare X in
thread {Delay 10000} X=99 end
{Browse start} {Browse X*X}
```

The multiplication `X*X` waits until `X` is bound. The first `Browse` immediately displays `start`. The second `Browse` waits for the multiplication, so it displays nothing yet. The `{Delay 10000}` call pauses for 10000 ms (i.e., 10 seconds). `X` is bound only after the delay continues. When `X` is bound, then the multiplication continues and the second browse displays 9801. The two operations `X=99` and `X*X` can be done in any order with any kind of delay; dataflow execution will always give the same result. The only effect a delay can have is to slow things down. For example:

```
declare X in
thread {Browse start} {Browse X*X} end
{Delay 10000} X=99
```

This behaves exactly as before: the browser displays 9801 after 10 seconds. This illustrates two nice properties of dataflow. First, calculations work correctly independent of how they are partitioned between threads. Second, calculations are patient: they do not signal errors, but simply wait.

Adding threads and delays to a program can radically change a program's appearance. But as long as the same operations are invoked with the same arguments, it does not change the program's results at all. This is the key property of dataflow concurrency. This is why dataflow concurrency gives most of the advantages of concurrency without the complexities that are usually associated with it. Dataflow concurrency is covered in chapter 4.

1.12 Explicit state

How can we let a function learn from its past? That is, we would like the function to have some kind of internal memory, which helps it do its job. Memory is needed for functions that can change their behavior and learn from their past. This kind of memory is called explicit state. Just like for concurrency, explicit state models an essential aspect of how the real world works. We would like to be able to do this in the system as well. Later in the book we will see deeper reasons for having explicit state (see chapter 6). For now, let us just see how it works.

For example, we would like to see how often the `FastPascal` function is used. Is there some way `FastPascal` can remember how many times it was called? We can do this by adding explicit state.

A memory cell

There are lots of ways to define explicit state. The simplest way is to define a single memory cell. This is a kind of box in which you can put any content. Many programming languages call this a "variable." We call it a "cell" to avoid confusion

with the variables we used before, which are more like mathematical variables, i.e., just shortcuts for values. There are three functions on cells: `NewCell` creates a new cell, `:=` (assignment) puts a new value in a cell, and `@` (access) gets the current value stored in the cell. Access and assignment are also called read and write. For example:

```
declare
C={NewCell 0}
C:=@C+1
{Browse @C}
```

This creates a cell `C` with initial content 0, adds one to the content, and displays it.

Adding memory to `FastPascal`

With a memory cell, we can let `FastPascal` count how many times it is called. First we create a cell outside of `FastPascal`. Then, inside of `FastPascal`, we add 1 to the cell's content. This gives the following:

```
declare
C={NewCell 0}
fun {FastPascal N}
   C:=@C+1
   {GenericPascal Add N}
end
```

(To keep it short, this definition uses `GenericPascal`.)

1.13 Objects

A function with internal memory is usually called an *object*. The extended version of `FastPascal` we defined in the previous section is an object. It turns out that objects are very useful beasts. Let us give another example. We will define a counter object. The counter has a cell that keeps track of the current count. The counter has two operations, `Bump` and `Read`, which we call its interface. `Bump` adds 1 and then returns the resulting count. `Read` just returns the count. Here is the definition:

```
declare
local C in
   C={NewCell 0}
   fun {Bump}
      C:=@C+1
      @C
   end
   fun {Read}
      @C
   end
end
```

The **`local`** statement declares a new variable `C` that is visible only up to the matching **`end`**. There is something special going on here: the cell is referenced

by a local variable, so it is completely invisible from the outside. This is called encapsulation. Encapsulation implies that users cannot mess with the counter's internals. We can guarantee that the counter will always work correctly no matter how it is used. This was not true for the extended `FastPascal` because anyone could look at and modify the cell.

It follows that as long as the interface to the counter object is the same, the user program does not need to know the implementation. The separation of interface and implementation is the essence of data abstraction. It can greatly simplify the user program. A program that uses a counter will work correctly for any implementation as long as the interface is the same. This property is called polymorphism. Data abstraction with encapsulation and polymorphism is covered in chapter 6 (see section 6.4).

We can bump the counter up:

```
{Browse {Bump}}
{Browse {Bump}}
```

What does this display? `Bump` can be used anywhere in a program to count how many times something happens. For example, `FastPascal` could use `Bump`:

```
declare
fun {FastPascal N}
   {Browse {Bump}}
   {GenericPascal Add N}
end
```

1.14 Classes

The last section defined one counter object. What if we need more than one counter? It would be nice to have a "factory" that can make as many counters as we need. Such a factory is called a *class*. Here is one way to define it:

```
declare
fun {NewCounter}
C Bump Read in
   C={NewCell 0}
   fun {Bump}
      C:=@C+1
      @C
   end
   fun {Read}
      @C
   end
   counter(bump:Bump read:Read)
end
```

`NewCounter` is a function that creates a new cell and returns new `Bump` and `Read` functions that use the cell. Returning functions as results of functions is another form of higher-order programming.

1.15 Nondeterminism and time

We have seen how to add concurrency and state to a program separately. What happens when a program has both? It turns out that having both at the same time is a tricky business, because the same program can give different results from one execution to the next. This is because the order in which threads access the state can change from one execution to the next. This variability is called nondeterminism. Nondeterminism exists because we lack knowledge of the exact time when each basic operation executes. If we would know the exact time, then there would be no nondeterminism. But we cannot know this time, simply because threads are independent. Since they know nothing of each other, they also do not know which instructions each has executed.

Nondeterminism by itself is not a problem; we already have it with concurrency. The difficulties occur if the nondeterminism shows up in the program, i.e., if it is observable. An observable nondeterminism is sometimes called a race condition. Here is an example:

```
declare
C={NewCell 0}
thread
   C:=1
end
thread
   C:=2
end
```

What is the content of `C` after this program executes? Figure 1.4 shows the two possible executions of this program. Depending on which one is done, the final cell content can be either 1 or 2. The problem is that we cannot say which. This is a simple case of observable nondeterminism. Things can get much trickier. For example, let us use a cell to hold a counter that can be incremented by several threads:

```
declare
C={NewCell 0}
thread I in
   I=@C
   C:=I+1
end
thread J in
   J=@C
   C:=J+1
end
```

What is the content of `C` after this program executes? It looks like each thread just adds 1 to the content, making it 2. But there is a surprise lurking: the final content can also be 1! How is this possible? Try to figure out why before continuing.

We need two operations on locks. First, we create a new lock by calling the function `NewLock`. Second, we define the lock's inside with the instruction **`lock`** `L` **`then ... end`**, where `L` is a lock. Now we can fix the cell counter:

```
declare
C={NewCell 0}
L={NewLock}
thread
   lock L then I in
      I=@C
      C:=I+1
   end
end
thread
   lock L then J in
      J=@C
      C:=J+1
   end
end
```

In this version, the final result is always 2. Both thread bodies have to be guarded by the same lock, otherwise the undesirable interleaving can still occur. Do you see why?

1.17 Where do we go from here?

This chapter has given a quick overview of many of the most important concepts in programming. The intuitions given here will serve you well in the chapters to come, when we define in a precise way the concepts and the computation models they are part of. This chapter has introduced the following computation models:

- Declarative model (chapters 2 and 3). Declarative programs define mathematical functions. They are the easiest to reason about and to test. The declarative model is important also because it contains many of the ideas that will be used in later, more expressive models.
- Concurrent declarative model (chapter 4). Adding dataflow concurrency gives a model that is still declarative but that allows a more flexible, incremental execution.
- Lazy declarative model (section 4.5). Adding laziness allows calculating with potentially infinite data structures. This is good for resource management and program structure.
- Stateful model (chapter 6). Adding explicit state allows writing programs whose behavior changes over time. This is good for program modularity. If written well, i.e., using encapsulation and invariants, these programs are almost as easy to reason about as declarative programs.
- Object-oriented model (chapter 7). Object-oriented programming is a programming style for stateful programming with data abstractions. It makes it easy to use

powerful techniques such as polymorphism and inheritance.

- Shared-state concurrent model (chapter 8). This model adds both concurrency and explicit state. If programmed carefully, using techniques for mastering interleaving such as monitors and transactions, this gives the advantages of both the stateful and concurrent models.

In addition to these models, the book covers many other useful models such as the declarative model with exceptions (section 2.7), the message-passing concurrent model (chapter 5), the relational model (chapter 9), and the specialized models of part II.

1.18 Exercises

1. *A calculator.* Section 1.1 uses the system as a calculator. Let us explore the possibilities:

   (a) Calculate the exact value of 2^{100} without using any new functions. Try to think of shortcuts to do it without having to type `2*2*2*...*2` with one hundred 2s. *Hint*: use variables to store intermediate results.

   (b) Calculate the exact value of 100! without using any new functions. Are there any possible shortcuts in this case?

2. *Calculating combinations.* Section 1.3 defines the function `Comb` to calculate combinations. This function is not very efficient because it might require calculating very large factorials. The purpose of this exercise is to write a more efficient version of `Comb`.

   (a) As a first step, use the following alternative definition to write a more efficient function:

   $$\binom{n}{k} = \frac{n \times (n-1) \times \cdots \times (n-k+1)}{k \times (k-1) \times \cdots \times 1}$$

   Calculate the numerator and denominator separately and then divide them. Make sure that the result is 1 when $k = 0$.

   (b) As a second step, use the following identity:

   $$\binom{n}{k} = \binom{n}{n-k}$$

   to increase efficiency even more. That is, if $k > n/2$, then do the calculation with $n - k$ instead of with k.

3. *Program correctness.* Section 1.6 explains the basic ideas of program correctness and applies them to show that the factorial function defined in section 1.3 is correct. In this exercise, apply the same ideas to the function `Pascal` of section 1.5 to show that it is correct.

4. *Program complexity.* What does section 1.7 say about programs whose time complexity is a high-order polynomial? Are they practical or not? What do you

think?

5. *Lazy evaluation.* Section 1.8 defines the lazy function `Ints` that lazily calculates an infinite list of integers. Let us define a function that calculates the sum of a list of integers:

```
fun {SumList L}
   case L of X|L1 then X+{SumList L1}
   else 0 end
end
```

What happens if we call `{SumList {Ints 0}}`? Is this a good idea?

6. *Higher-order programming.* Section 1.9 explains how to use higher-order programming to calculate variations on Pascal's triangle. The purpose of this exercise is to explore these variations.

(a) Calculate individual rows using subtraction, multiplication, and other operations. Why does using multiplication give a triangle with all zeros? Try the following kind of multiplication instead:

```
fun {Mul1 X Y} (X+1)*(Y+1) end
```

What does the 10th row look like when calculated with `Mul1`?

(b) The following loop instruction will calculate and display 10 rows at a time:

```
for I in 1..10 do {Browse {GenericPascal Op I}} end
```

Use this loop instruction to make it easier to explore the variations.

7. *Explicit state.* This exercise compares variables and cells. We give two code fragments. The first uses variables:

```
local X in
   X=23
   local X in
      X=44
   end
   {Browse X}
end
```

The second uses a cell:

```
local X in
   X={NewCell 23}
   X:=44
   {Browse @X}
end
```

In the first, the identifier `X` refers to two different variables. In the second, `X` refers to a cell. What does `Browse` display in each fragment? Explain.

8. *Explicit state and functions.* This exercise investigates how to use cells together with functions. Let us define a function `{Accumulate N}` that accumulates all its inputs, i.e., it adds together all the arguments of all calls. Here is an example:

```
{Browse {Accumulate 5}}
{Browse {Accumulate 100}}
{Browse {Accumulate 45}}
```

This should display 5, 105, and 150, assuming that the accumulator contains zero at the start. Here is a wrong way to write `Accumulate`:

```
declare
fun {Accumulate N}
Acc in
   Acc={NewCell 0}
   Acc:=@Acc+N
   @Acc
end
```

What is wrong with this definition? How would you correct it?

9. *Memory store.* This exercise investigates another way of introducing state: a memory store. The memory store can be used to make an improved version of `FastPascal` that remembers previously calculated rows.

(a) A memory store is similar to the memory of a computer. It has a series of memory cells, numbered from 1 up to the maximum used so far. There are four functions on memory stores: `NewStore` creates a new store, `Put` puts a new value in a memory cell, `Get` gets the current value stored in a memory cell, and `Size` gives the number of cells used so far. For example:

```
declare
S={NewStore}
{Put S 2 [22 33]}
{Browse {Get S 2}}
{Browse {Size S}}
```

This stores `[22 33]` in memory cell 2, displays `[22 33]`, and then displays 1. Load into the Mozart system the memory store as defined in the supplements file on the book's Web site. Then use the interactive interface to understand how the store works.

(b) Now use the memory store to write an improved version of `FastPascal`, called `FasterPascal`, that remembers previously calculated rows. If a call asks for one of these rows, then the function can return it directly without having to recalculate it. This technique is sometimes called memoization since the function makes a "memo" of its previous work. This improves its performance. Here's how it works:

- First make a store S available to `FasterPascal`.
- For the call `{FasterPascal N}`, let m be the number of rows stored in S, i.e., rows 1 up to m are in S.
- If $n > m$, then compute rows $m + 1$ up to n and store them in S.
- Return the nth row by looking it up in S.

Viewed from the outside, `FasterPascal` behaves identically to `FastPascal` except that it is faster.

(c) We have given the memory store as a library. It turns out that the memory store can be defined by using a memory cell. We outline how it can be done and you can write the definitions. The cell holds the store contents as a list of

I GENERAL COMPUTATION MODELS

2 Declarative Computation Model

Non sunt multiplicanda entia praeter necessitatem.
Do not multiply entities beyond necessity.
– Ockham's razor, after William of Ockham (1285?–1347/49)

Programming encompasses three things:

- First, a computation model, which is a formal system that defines a language and how sentences of the language (e.g., expressions and statements) are executed by an abstract machine. For this book, we are interested in computation models that are useful and intuitive for programmers. This will become clearer when we define the first one later in this chapter.
- Second, a set of programming techniques and design principles used to write programs in the language of the computation model. We will sometimes call this a programming model. A programming model is always built on top of a computation model.
- Third, a set of reasoning techniques to let you reason about programs, to increase confidence that they behave correctly, and to calculate their efficiency.

The above definition of computation model is very general. Not all computation models defined in this way will be useful for programmers. What is a reasonable computation model? Intuitively, we will say that a reasonable model is one that can be used to solve many problems, that has straightforward and practical reasoning techniques, and that can be implemented efficiently. We will have more to say about this question later on. The first and simplest computation model we will study is declarative programming. For now, we define this as evaluating functions over partial data structures. This is sometimes called stateless programming, as opposed to stateful programming (also called imperative programming) which is explained in chapter 6.

The declarative model of this chapter is one of the most fundamental computation models. It encompasses the core ideas of the two main declarative paradigms, namely functional and logic programming. It encompasses programming with functions over complete values, as in Scheme and Standard ML. It also encompasses deterministic logic programming, as in Prolog when search is not used. And finally, it can be made concurrent without losing its good properties (see chapter 4).

Declarative programming is a rich area—it has most of the ideas of the more

expressive computation models, at least in embryonic form. We therefore present it in two chapters. This chapter defines the computation model and a practical language based on it. The next chapter, chapter 3, gives the programming techniques of this language. Later chapters enrich the basic model with many concepts. Some of the most important are exception handling, concurrency, components (for programming in the large), capabilities (for encapsulation and security), and state (leading to objects and classes). In the context of concurrency, we will talk about dataflow, lazy execution, message passing, active objects, monitors, and transactions. We will also talk about user interface design, distribution (including fault tolerance), and constraints (including search).

Structure of the chapter

The chapter consists of eight sections:

- Section 2.1 explains how to define the syntax and semantics of practical programming languages. Syntax is defined by a context-free grammar extended with language constraints. Semantics is defined in two steps: by translating a practical language into a simple kernel language and then giving the semantics of the kernel language. These techniques will be used throughout the book. This chapter uses them to define the declarative computation model.
- The next three sections define the syntax and semantics of the declarative model:
  - Section 2.2 gives the data structures: the single-assignment store and its contents, partial values and dataflow variables.
  - Section 2.3 defines the kernel language syntax.
  - Section 2.4 defines the kernel language semantics in terms of a simple abstract machine. The semantics is designed to be intuitive and to permit straightforward reasoning about correctness and complexity.
- Section 2.5 uses the abstract machine to explore the memory behavior of computations. We look at last call optimization and the concept of memory life cycle.
- Section 2.6 defines a practical programming language on top of the kernel language.
- Section 2.7 extends the declarative model with exception handling, which allows programs to handle unpredictable and exceptional situations.
- Section 2.8 gives a few advanced topics to let interested readers deepen their understanding of the model.

2.1 Defining practical programming languages

Programming languages are much simpler than natural languages, but they can still have a surprisingly rich syntax, set of abstractions, and libraries. This is especially

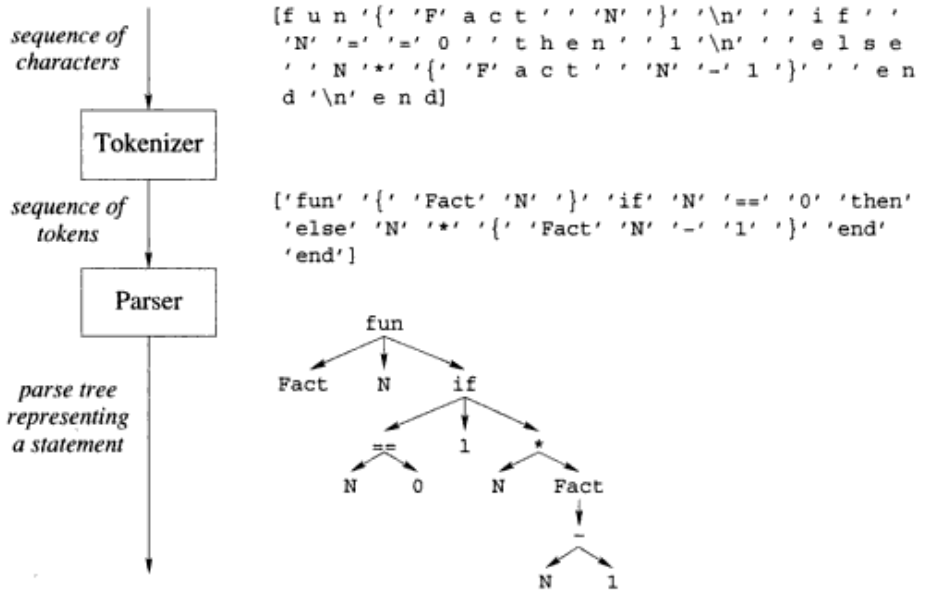


Figure 2.1: From characters to statements.

true for languages that are used to solve real-world problems, which we call practical languages. A practical language is like the toolbox of an experienced mechanic: there are many different tools for many different purposes and all tools are there for a reason.

This section sets the stage for the rest of the book by explaining how we will present the syntax ("grammar") and semantics ("meaning") of practical programming languages. With this foundation we will be ready to present the first computation model of the book, namely the declarative computation model. We will continue to use these techniques throughout the book to define computation models.

2.1.1 Language syntax

The syntax of a language defines what are the legal programs, i.e., programs that can be successfully executed. At this stage we do not care what the programs are actually doing. That is semantics and will be handled in section 2.1.2.

Grammars

A grammar is a set of rules that defines how to make 'sentences' out of 'words'. Grammars can be used for natural languages, like English or Swedish, as well as for artificial languages, like programming languages. For programming languages, 'sentences' are usually called 'statements' and 'words' are usually called 'tokens'. Just as words are made of letters, tokens are made of characters. This gives us two

levels of structure:

statement ('sentence')	=	sequence of tokens ('words')
token ('word')	=	sequence of characters ('letters')

Grammars are useful both for defining statements and tokens. Figure 2.1 gives an example to show how character input is transformed into a statement. The example in the figure is the definition of `Fact`:

```
fun {Fact N}
   if N==0 then 1
   else N*{Fact N-1} end
end
```

The input is a sequence of characters, where `' '` represents the space and `'\n'` represents the newline. This is first transformed into a sequence of tokens and subsequently into a parse tree. The syntax of both sequences in the figure is compatible with the list syntax we use throughout the book. Whereas the sequences are "flat," the parse tree shows the structure of the statement. A program that accepts a sequence of characters and returns a sequence of tokens is called a tokenizer or lexical analyzer. A program that accepts a sequence of tokens and returns a parse tree is called a parser.

Extended Backus-Naur Form

One of the most common notations for defining grammars is called Extended Backus-Naur Form (EBNF), after its inventors John Backus and Peter Naur. The EBNF notation distinguishes terminal symbols and nonterminal symbols. A terminal symbol is simply a token. A nonterminal symbol represents a sequence of tokens. The nonterminal is defined by means of a grammar rule, which shows how to expand it into tokens. For example, the following rule defines the nonterminal ⟨digit⟩:

⟨digit⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

It says that ⟨digit⟩ represents one of the ten tokens 0, 1, ..., 9. The symbol "|" is read as "or"; it means to pick one of the alternatives. Grammar rules can themselves refer to other nonterminals. For example, we can define a nonterminal ⟨int⟩ that defines how to write positive integers:

⟨int⟩ ::= ⟨digit⟩ { ⟨digit⟩ }

This rule says that an integer is a digit followed by any number of digits, including none. The braces "{ ... }" mean to repeat whatever is inside any number of times, including none.

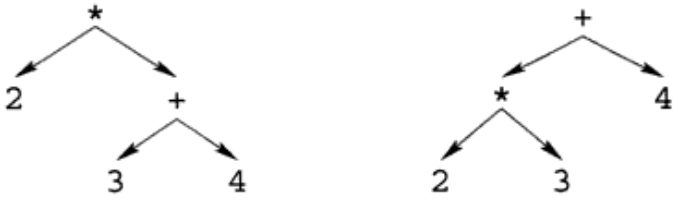


Figure 2.3: Ambiguity in a context-free grammar.

conditions imposed by the language. The context-free grammar is kept instead of some more expressive notation because it is easy to read and understand. It has an important locality property: a nonterminal symbol can be understood by examining only the rules needed to define it; the (possibly much more numerous) rules that use it can be ignored. The context-free grammar is corrected by imposing a set of extra conditions, like the declare-before-use restriction on variables. Taking these conditions into account gives a context-sensitive grammar.

Ambiguity

Context-free grammars can be ambiguous, i.e., there can be several parse trees that correspond to a given token sequence. For example, here is a simple grammar for arithmetic expressions with addition and multiplication:

⟨exp⟩ ::= ⟨int⟩ | ⟨exp⟩ ⟨op⟩ ⟨exp⟩
⟨op⟩ ::= + | *

The expression `2*3+4` has two parse trees, depending on how the two occurrences of ⟨exp⟩ are read. Figure 2.3 shows the two trees. In one tree, the first ⟨exp⟩ is `2` and the second ⟨exp⟩ is `3+4`. In the other tree, they are `2*3` and `4`, respectively.

Ambiguity is usually an undesirable property of a grammar since it is unclear exactly what program is being written. In the expression `2*3+4`, the two parse trees give different results when evaluating the expression: one gives 14 (the result of computing 2*(3+4)) and the other gives 10 (the result of computing (2*3)+4). Sometimes the grammar rules can be rewritten to remove the ambiguity, but this can make the rules more complicated. A more convenient approach is to add extra conditions. These conditions restrict the parser so that only one parse tree is possible. We say that they disambiguate the grammar.

For expressions with binary operators such as the arithmetic expressions given above, the usual approach is to add two conditions, precedence and associativity:

- Precedence is a condition on an expression with different operators, like `2*3+4`. Each operator is given a precedence level. Operators with high precedences are put as deep in the parse tree as possible, i.e., as far away from the root as possible. If `*` has higher precedence than `+`, then the parse tree `(2*3)+4` is chosen over the

⟨statement⟩ ::= **if** ⟨expression⟩ **then** ⟨statement⟩
{ **elseif** ⟨expression⟩ **then** ⟨statement⟩ }
[**else** ⟨statement⟩] **end** | ...
⟨expression⟩ ::= ´[´ { ⟨expression⟩ }+ ´]´ | ...
⟨label⟩ ::= **unit** | **true** | **false** | ⟨variable⟩ | ⟨atom⟩

The first rule defines the **if** statement. There is an optional sequence of **elseif** clauses, i.e., there can be any number of occurrences including zero. This is denoted by the braces { ... }. This is followed by an optional **else** clause, i.e., it can occur zero or one times. This is denoted by the brackets [...]. The second rule defines the syntax of explicit lists. They must have at least one element, e.g., `[5 6 7]` is valid but `[ ]` is not (note the space that separates the `[` and the `]`). This is denoted by { ... }+. The third rule defines the syntax of record labels. The third rule is a complete definition since there are no three dots "...". There are five possibilities and no more will ever be given.

2.1.2 Language semantics

The semantics of a language defines what a program does when it executes. Ideally, the semantics should be defined in a simple mathematical structure that lets us reason about the program (including its correctness, execution time, and memory use) without introducing any irrelevant details. Can we achieve this for a practical language without making the semantics too complicated? The technique we use, which we call the kernel language approach, gives an affirmative answer to this question.

Modern programming languages have evolved through more than five decades of experience in constructing programmed solutions to complex, real-world problems.[1] Modern programs can be quite complex, reaching sizes measured in millions of lines of code, written by large teams of human programmers over many years. In our view, languages that scale to this level of complexity are successful in part because they model some essential aspects of how to construct complex programs. In this sense, these languages are not just arbitrary constructions of the human mind. We would therefore like to understand them in a scientific way, i.e., by explaining their behavior in terms of a simple underlying model. This is the deep motivation behind the kernel language approach.

1. The figure of five decades is somewhat arbitrary. We measure it from the first working stored-program computer, the Manchester Mark I. According to lab documents, it ran its first program on June 21, 1948 [197].

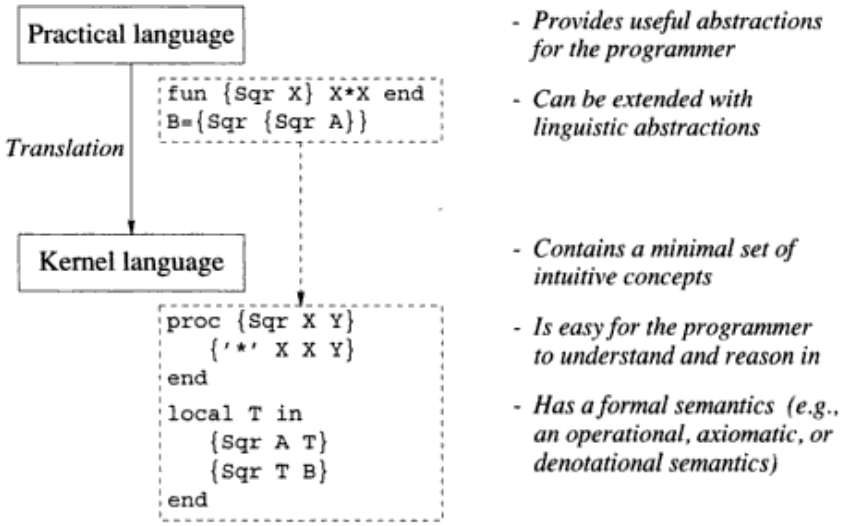


Figure 2.4: The kernel language approach to semantics.

The kernel language approach

This book uses the kernel language approach to define the semantics of programming languages. In this approach, all language constructs are defined in terms of translations into a core language known as the kernel language. The kernel language approach consists of two parts (see figure 2.4):

- First, define a very simple language, called the kernel language. This language should be easy to reason in and be faithful to the space and time efficiency of the implementation. The kernel language and the data structures it manipulates together form the kernel computation model.
- Second, define a translation scheme from the full programming language to the kernel language. Each grammatical construct in the full language is translated into the kernel language. The translation should be as simple as possible. There are two kinds of translation, namely linguistic abstraction and syntactic sugar. Both are explained below.

The kernel language approach is used throughout the book. Each computation model has its kernel language, which builds on its predecessor by adding one new concept. The first kernel language, which is presented in this chapter, is called the declarative kernel language. Many other kernel languages are presented later on in the book.

Formal semantics

The kernel language approach lets us define the semantics of the kernel language in any way we want. There are four widely used approaches to language semantics:

- An operational semantics shows how a statement executes in terms of an abstract machine. This approach always works well, since at the end of the day all languages execute on a computer.
- An axiomatic semantics defines a statement's semantics as the relation between the input state (the situation before executing the statement) and the output state (the situation after executing the statement). This relation is given as a logical assertion. This is a good way to reason about statement sequences, since the output assertion of each statement is the input assertion of the next. It therefore works well with stateful models, since a state is a sequence of values. Section 6.6 gives an axiomatic semantics of chapter 6's stateful model.
- A denotational semantics defines a statement as a function over an abstract domain. This works well for declarative models, but can be applied to other models as well. It gets complicated when applied to concurrent languages. Sections 2.8.1 and 4.9.2 explain functional programming, which is particularly close to denotational semantics.
- A logical semantics defines a statement as a model of a logical theory. This works well for declarative and relational computation models, but is hard to apply to other models. Section 9.3 gives a logical semantics of the declarative and relational computation models.

Much of the theory underlying these different semantics is of interest primarily to mathematicians, not to programmers. It is outside the scope of the book to give this theory. The principal formal semantics we give in the book is an operational semantics. We define it for each computation model. It is detailed enough to be useful for reasoning about correctness and complexity yet abstract enough to avoid irrelevant clutter. Chapter 13 collects all these operational semantics into a single formalism with a compact and readable notation.

Throughout the book, we give an informal semantics for every new language construct and we often reason informally about programs. These informal presentations are always based on the operational semantics.

Linguistic abstraction

Both programming languages and natural languages can evolve to meet their needs. When using a programming language, at some point we may feel the need to extend the language, i.e., to add a new linguistic construct. For example, the declarative model of this chapter has no looping constructs. Section 3.6.3 defines a **`for`** construct to express certain kinds of loops that are useful for writing declarative programs. The new construct is both an abstraction and an addition to the language syntax.

We therefore call it a linguistic abstraction. A practical programming language contains many linguistic abstractions.

There are two phases to defining a linguistic abstraction. First, define a new grammatical construct. Second, define its translation into the kernel language. The kernel language is not changed. This book gives many examples of useful linguistic abstractions, e.g., functions (**fun**), loops (**for**), lazy functions (**fun lazy**), classes (**class**), reentrant locks (**lock**), and others.[2] Some of these are part of the Mozart system. The others can be added to Mozart with the **gump** parser-generator tool [117]. Using this tool is beyond the scope of the book.

Some languages have facilities for programming linguistic abstractions directly in the language. A simple yet powerful example is the Lisp macro. A Lisp macro resembles a function that generates Lisp code when executed. Partly because of Lisp's simple syntax, macros have been extraordinarily successful in Lisp and its successors. Lisp has built-in support for macros, such as quote (turning a program expression into a data structure) and backquote (doing the inverse, inside a quoted structure). For a detailed discussion of Lisp macros and related ideas we refer the reader to any good book on Lisp [72, 200].

A simple example of a linguistic abstraction is the function, which uses the keyword **fun**. This is explained in section 2.6.2. We have already programmed with functions in chapter 1. But the kernel language of this chapter only has procedures. Procedures are used since all arguments are explicit and there can be multiple outputs. There are other, deeper reasons for choosing procedures which are explained later in this chapter. Because functions are so useful, though, we add them as a linguistic abstraction.

We define a syntax for both function definitions and function calls, and a translation into procedure definitions and procedure calls. The translation lets us answer all questions about function calls. For example, what does `{F1 {F2 X} {F3 Y}}` mean exactly (nested function calls)? Is the order of these function calls defined? If so, what is the order? There are many possibilities. Some languages leave the order of argument evaluation unspecified, but assume that a function's arguments are evaluated before the function. Other languages assume that an argument is evaluated when and if its result is needed, not before. So even as simple a thing as nested function calls does not necessarily have an obvious semantics. The translation makes it clear what the semantics is.

Linguistic abstractions are useful for more than just increasing the expressiveness of a program. They can also improve other properties such as correctness, security, and efficiency. By hiding the abstraction's implementation from the programmer, the linguistic support makes it impossible to use the abstraction in the wrong way. The compiler can use this information to give more efficient code.

---

2. Logic gates (**gate**) for circuit descriptions, mailboxes (**receive**) for message-passing concurrency, and currying and list comprehensions as in modern functional languages, cf. Haskell.

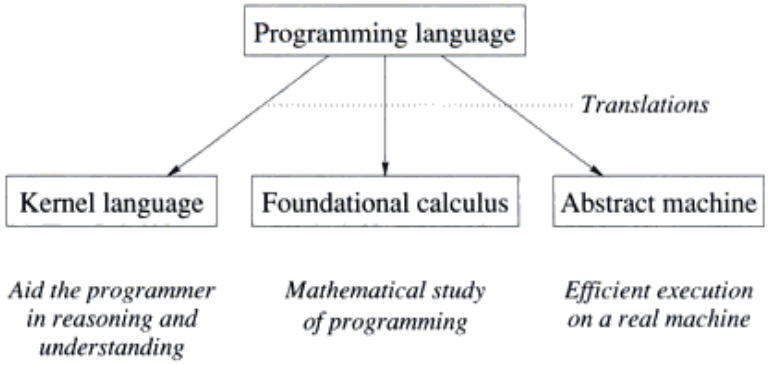


Figure 2.5: Translation approaches to language semantics.

- The kernel language approach, used throughout the book, is intended for the programmer. Its concepts correspond directly to programming concepts.
- The foundational approach is intended for the mathematician. Examples are the Turing machine, the λ calculus (underlying functional programming), first-order logic (underlying logic programming), and the π calculus (to model concurrency). Because these calculi are intended for formal mathematical study, they have as few elements as possible.
- The abstract machine approach is intended for the implementor. Programs are translated into an idealized machine, which is traditionally called an *abstract machine* or a *virtual machine*.[3] It is relatively easy to translate idealized machine code into real machine code.

Because we focus on practical programming techniques, the book uses only the kernel language approach. The other two approaches have the problem that any realistic program written in them is cluttered with technical details about language mechanisms. The kernel language approach avoids this clutter by a careful choice of concepts.

The interpreter approach

An alternative to the translation approach is the interpreter approach. The language semantics is defined by giving an interpreter for the language. New language features

3. Strictly speaking, a virtual machine is a software emulation of a real machine, running on the real machine, that is almost as efficient as the real machine. It achieves this efficiency by executing most virtual instructions directly as real instructions. The concept was pioneered by IBM in the early 1960s in the VM operating system. Because of the success of Java, which uses the term "virtual machine," modern usage also uses the term virtual machine in the sense of abstract machine.

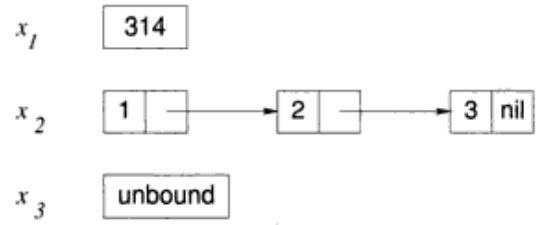


Figure 2.7: Two of the variables are bound to values.

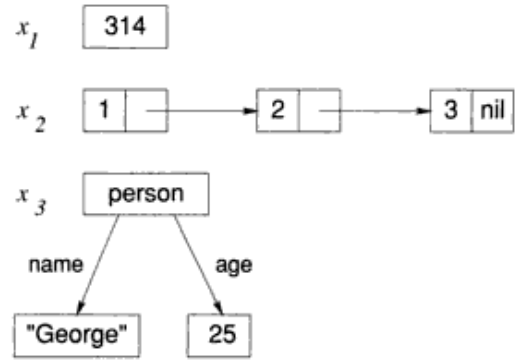


Figure 2.8: A value store: all variables are bound to values.

calculations as if it were the value. Doing the operation $x + y$ is the same as doing $11 + 22$, if the store is $\{x = 11, y = 22\}$.

2.2.2 Value store

A store where all variables are bound to values is called a value store. Another way to say this is that a value store is a persistent mapping from variables to values. A value is a mathematical constant. For example, the integer 314 is a value. Values can also be compound entities, i.e., entities that contain one or more other values. For example, the list `[1 2 3]` and the record `person(name:"George" age:25)` are values. Figure 2.8 shows a value store where x_1 is bound to the integer 314, x_2 is bound to the list `[1 2 3]`, and x_3 is bound to the record `person(name:"George" age:25)`. Functional languages such as Standard ML, Haskell, and Scheme get by with a value store since they compute functions on values. (Object-oriented languages such as Smalltalk, C++, and Java need a cell store, which consists of cells whose content can be modified.)

At this point, a reader with some programming experience may wonder why we are introducing a single-assignment store, when other languages get by with

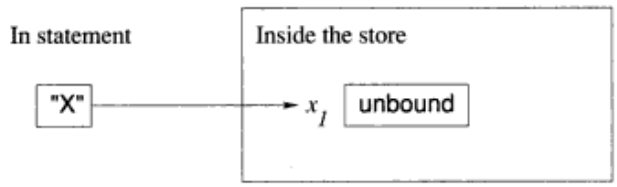


Figure 2.9: A variable identifier referring to an unbound variable.

a value store or a cell store. There are many reasons. A first reason is that we want to compute with partial values. For example, a procedure can return an output by binding an unbound variable argument. A second reason is declarative concurrency, which is the subject of chapter 4. It is possible because of the single-assignment store. A third reason is that a single-assignment store is needed for relational (logic) programming and constraint programming. Other reasons having to do with efficiency (e.g., tail recursion and difference lists) will become clear in the next chapter.

2.2.3 Value creation

The basic operation on a store is binding a variable to a newly created value. We will write this as x_i=*value*. Here x_i refers directly to a variable in the store (it is not the variable's textual name in a program!) and *value* refers to a value, e.g., 314 or `[1 2 3]`. For example, figure 2.7 shows the store of figure 2.6 after the two bindings:

$x_1 = 314$
$x_2 =$ `[1 2 3]`

The single-assignment operation x_i=*value* constructs *value* in the store and then binds the variable x_i to this value. If the variable is already bound, the operation will test whether the two values are compatible. If they are not compatible, an error is signaled (using the exception-handling mechanism; see section 2.7).

2.2.4 Variable identifiers

So far, we have looked at a store that contains variables and values, i.e., store entities, with which calculations can be done. It would be nice if we could refer to a store entity from outside the store. This is the role of variable identifiers. A variable identifier is a textual name that refers to a store entity from outside the store. The mapping from variable identifiers to store entities is called an environment.

The variable names in program source code are in fact variable identifiers. For example, figure 2.9 has an identifier "X" (the capital letter X) that refers to the store variable x_1. This corresponds to the environment $\{\mathtt{X} \rightarrow x_1\}$. To talk about

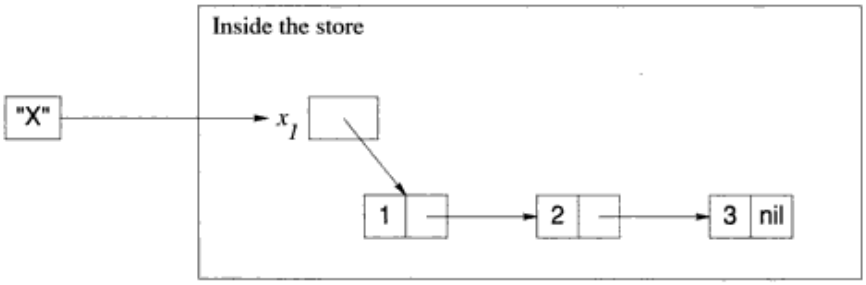


Figure 2.10: A variable identifier referring to a bound variable.

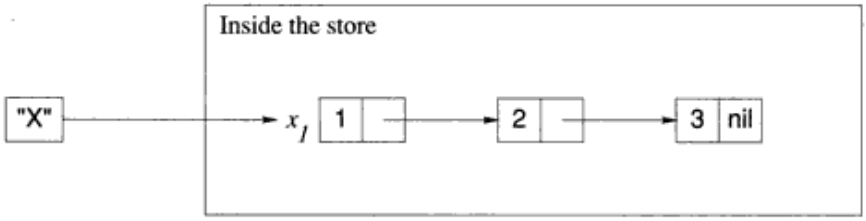


Figure 2.11: A variable identifier referring to a value.

any identifier, we will use the notation ⟨x⟩. The environment $\{\langle x\rangle \rightarrow x_1\}$ is the same as before, if ⟨x⟩ represents X. As we will see later, variable identifiers and their corresponding store entities are added to the environment by the **`local`** and **`declare`** statements.

2.2.5 Value creation with identifiers

Once bound, a variable is indistinguishable from its value. Figure 2.10 shows what happens when x_1 is bound to `[1 2 3]` in figure 2.9. With the variable identifier `X`, we can write the binding as `X=[1 2 3]`. This is the text a programmer would write to express the binding. We can also use the notation ⟨x⟩`=[1 2 3]` if we want to be able to talk about any identifier. To make this notation legal in a program, ⟨x⟩ has to be replaced by an identifier.

The equality sign "=" refers to the bind operation. After the bind completes, the identifier "X" still refers to x_1, which is now bound to `[1 2 3]`. This is indistinguishable from figure 2.11, where `X` refers directly to `[1 2 3]`. Following the links of bound variables to get the value is called dereferencing. It is invisible to the programmer.

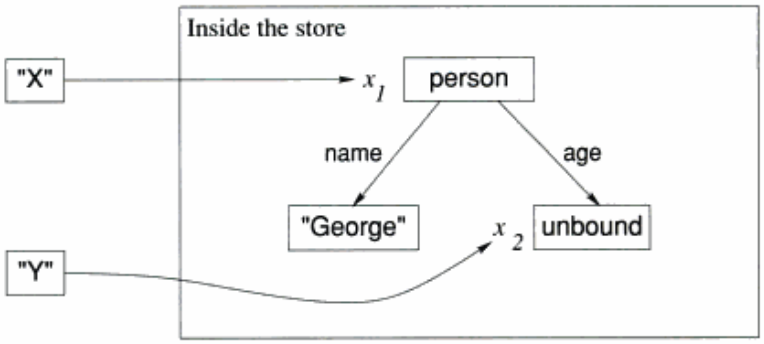


Figure 2.12: A partial value.

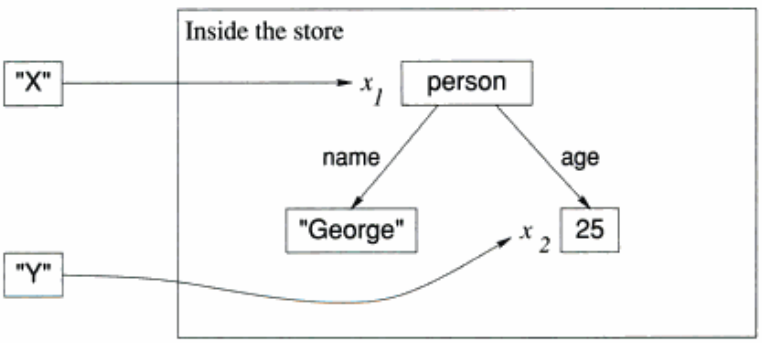


Figure 2.13: A partial value with no unbound variables, i.e., a complete value.

2.2.6 Partial values

A partial value is a data structure that may contain unbound variables. Figure 2.12 shows the record `person(name:"George" age:`x_2`)`, referred to by the identifier `X`. This is a partial value because it contains the unbound variable x_2. The identifier `Y` refers to x_2. Figure 2.13 shows the situation after x_2 is bound to 25 (through the bind operation `Y=25`). Now x_1 is a partial value with no unbound variables, which we call a complete value. A declarative variable can be bound to several partial values, as long as they are compatible with each other. We say a set of partial values is compatible if the unbound variables in them can be bound in such a way as to make them all equal. For example, `person(age:25)` and `person(age:`x`)` are compatible (because x can be bound to 25), but `person(age:25)` and `person(age:26)` are not.

variable use error. Some languages create and bind variables in one step, so that use errors cannot occur. This is the case for functional programming languages. Other languages allow creating and binding to be separate. Then we have the following possibilities when there is a use error:

1. Execution continues and no error message is given. The variable's content is undefined, i.e. it is "garbage": whatever is found in memory. This is what C++ does.

2. Execution continues and no error message is given. The variable is initialized to a default value when it is declared, e.g., to 0 for an integer. This is what Java does for fields in objects and data structures, such as arrays. The default value depends on the type.

3. Execution stops with an error message (or an exception is raised). This is what Prolog does for arithmetic operations.

4. Execution is not possible because the compiler detects that there is an execution path to the variable's use that does not initialize it. This is what Java does for local variables.

5. Execution waits until the variable is bound and then continues. This is what Oz does, to support dataflow programming.

These cases are listed in increasing order of niceness. The first case is very bad, since different executions of the same program can give different results. What's more, since the existence of the error is not signaled, the programmer is not even aware when this happens. The second case is somewhat better. If the program has a use error, then at least it will always give the same result, even if it is a wrong one. Again the programmer is not made aware of the error's existence.

The third and fourth cases can be reasonable in certain situations. In both cases, a program with a use error will signal this fact, either during execution or at compile time. This is reasonable in a sequential system, since there really is an error. The third case is unreasonable in a concurrent system, since the result becomes nondeterministic: depending on the execution timing, sometimes an error is signaled and sometimes not.

In the fifth case, the program will wait until the variable is bound and then continue. The computation models of the book use the fifth case. This is unreasonable in a sequential system, since the program will wait forever. It is reasonable in a concurrent system, where it could be part of normal operation that some other thread binds the variable. The fifth case introduces a new kind of program error, namely a suspension that waits forever. For example, if a variable name is misspelled then it will never be bound. A good debugger should detect when this occurs.

Declarative variables that cause the program to wait until they are bound are called *dataflow variables*. The declarative model uses dataflow variables because they are tremendously useful in concurrent programming, i.e., for programs with activities that run independently. If we do two concurrent operations, say `A=23` and `B=A+1`, then with the fifth case this will always run correctly and give the answer

⟨v⟩	::=	⟨number⟩ | ⟨record⟩ | ⟨procedure⟩
⟨number⟩	::=	⟨int⟩ | ⟨float⟩
⟨record⟩, ⟨pattern⟩	::=	⟨literal⟩
	|	⟨literal⟩(⟨feature⟩$_1$: ⟨x⟩$_1$ $\cdots$ ⟨feature⟩$_n$: ⟨x⟩$_n$)
⟨procedure⟩	::=	**proc** { $ ⟨x⟩$_1$ $\cdots$ ⟨x⟩$_n$ } ⟨s⟩ **end**
⟨literal⟩	::=	⟨atom⟩ | ⟨bool⟩
⟨feature⟩	::=	⟨atom⟩ | ⟨bool⟩ | ⟨int⟩
⟨bool⟩	::=	**true** | **false**

Table 2.2: Value expressions in the declarative kernel language.

that all variable-variable bindings are written as explicit kernel operations.

Variable identifier syntax

Table 2.1 uses the nonterminals ⟨x⟩ and ⟨y⟩ to denote a variable identifier. We will also use ⟨z⟩ to denote identifiers. There are two ways to write a variable identifier:

- An uppercase letter followed by zero or more alphanumeric characters (letters or digits or underscores), e.g., `X`, `X1`, or `ThisIsALongVariable_IsntIt`.
- Any sequence of printable characters enclosed within ` (backquote) characters, e.g., `` `this is a 25$\variable!` ``.

A precise definition of identifier syntax is given in appendix C. All newly declared variables are unbound before any statement is executed. All variable identifiers must be declared explicitly.

2.3.2 Values and types

A type or data type is a set of values together with a set of operations on those values. A value is "of a type" if it is in the type's set. The declarative model is typed in the sense that it has a well-defined set of types, called basic types. For example, programs can calculate with integers or with records, which are all of integer type or record type, respectively. Any attempt to use an operation with values of the wrong type is detected by the system and will raise an error condition (see section 2.7). The model imposes no other restrictions on the use of types.

Because all uses of types are checked, it is not possible for a program to behave outside of the model, e.g., to crash because of undefined operations on its internal data structures. It is still possible for a program to raise an error condition, e.g., by dividing by zero. In the declarative model, a program that raises an error condition will terminate immediately. There is nothing in the model to handle errors. In section 2.7 we extend the declarative model with a new concept, exceptions, to

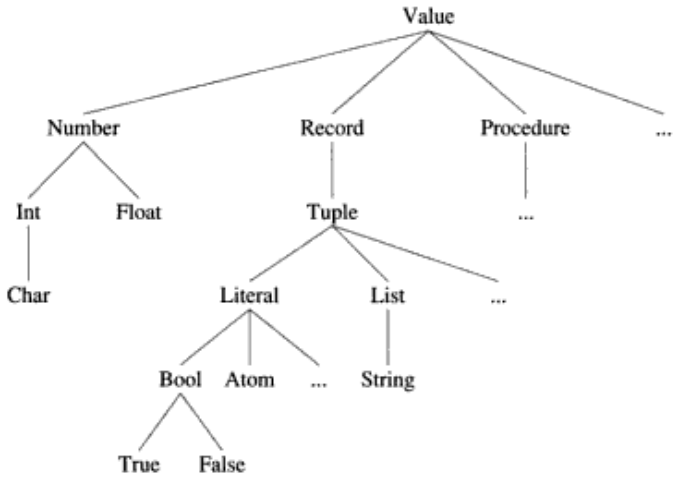


Figure 2.16: The type hierarchy of the declarative model.

handle errors. In the extended model, type errors can be handled within the model.

In addition to basic types, programs can define their own types. These are called abstract data types, or ADTs. Chapter 3 and later chapters show how to define ADTs. There are other kinds of data abstraction in addition to ADTs. Section 6.4 gives an overview of the different possibilities.

Basic types

The basic types of the declarative model are numbers (integers and floats), records (including atoms, booleans, tuples, lists, and strings), and procedures. Table 2.2 gives their syntax. The nonterminal ⟨v⟩ denotes a partially constructed value. Later in the book we will see other basic types, including chunks, functors, cells, dictionaries, arrays, ports, classes, and objects. Some of these are explained in appendix B.

Dynamic typing

There are two basic approaches to typing, namely dynamic and static typing. In static typing, all variable types are known at compile time. In dynamic typing, the variable type is known only when the variable is bound. The declarative model is dynamically typed. The compiler tries to verify that all operations use values of the correct type. But because of dynamic typing, some type checks are necessarily left for run time.

The type hierarchy

The basic types of the declarative model can be classified into a hierarchy. Figure 2.16 shows this hierarchy, where each node denotes a type. The hierarchy is ordered by set inclusion, i.e., all values of a node's type are also values of the parent node's type. For example, all tuples are records and all lists are tuples. This implies that all operations of a type are also legal for a subtype, e.g., all list operations work also for strings. Later on in the book we extend this hierarchy. For example, literals can be either atoms (explained below) or another kind of constant called names (see section 3.7.5). The parts where the hierarchy is incomplete are given as "...".

2.3.3 Basic types

We give some examples of the basic types and how to write them. See appendix B for more complete information.

- *Numbers.* Numbers are either integers or floating point numbers. Examples of integers are `314`, `0`, and `~10` (minus 10). Note that the minus sign is written with a tilde "~". Examples of floating point numbers are `1.0`, `3.4`, `2.0e2`, and `~2.0E~2`.
- *Atoms.* An atom is a kind of symbolic constant that can be used as a single element in calculations. There are several different ways to write atoms. An atom can be written as a sequence of characters starting with a lowercase letter followed by any number of alphanumeric characters. An atom can also be written as any sequence of printable characters enclosed in single quotes. Examples of atoms are `a_person`, `donkeyKong3`, and `'#### hello ####'`.
- *Booleans.* A boolean is either the symbol **`true`** or the symbol **`false`**.
- *Records.* A record is a compound data structure. It consists of a label followed by a set of pairs of features and variable identifiers. Features can be atoms, integers, or booleans. Examples of records are `person(age:X1 name:X2)` (with features `age` and `name`), `person(1:X1 2:X2)`, `'|'(1:H 2:T)`, `'#'(1:H 2:T)`, `nil`, and `person`. An atom is a record with no features.
- *Tuples.* A tuple is a record whose features are consecutive integers starting from 1. The features do not have to be written in this case. Examples of tuples are `person(1:X1 2:X2)` and `person(X1 X2)`, both of which mean the same.
- *Lists.* A list is either the atom `nil` or the tuple `'|'(H T)` (label is vertical bar), where `T` is either unbound or bound to a list. This tuple is called a list pair or a cons. There is syntactic sugar for lists:
  - The `'|'` label can be written as an infix operator, so that `H|T` means the same as `'|'(H T)`.
  - The `'|'` operator associates to the right, so that `1|2|3|nil` means the same as `1|(2|(3|nil))`.

- Lists that end in `nil` can be written with brackets `[ ... ]`, so that `[1 2 3]` means the same as `1|2|3|nil`. These lists are called complete lists.

- *Strings.* A string is a list of character codes. Strings can be written with double quotes, so that `"E=mc^2"` means the same as `[69 61 109 99 94 50]`.

- *Procedures.* A procedure is a value of the procedure type. The statement

⟨x⟩ = **proc** {$ ⟨y⟩$_1$ ··· ⟨y⟩$_n$} ⟨s⟩ **end**

binds ⟨x⟩ to a new procedure value. That is, it simply declares a new procedure. The `$` indicates that the procedure value is anonymous, i.e., created without being bound to an identifier. There is a syntactic shortcut that is more familiar:

proc {⟨x⟩ ⟨y⟩$_1$ ··· ⟨y⟩$_n$} ⟨s⟩ **end**

The `$` is replaced by the identifier ⟨x⟩. This creates the procedure value and immediately tries to bind it to ⟨x⟩. This shortcut is perhaps easier to read, but it blurs the distinction between creating the value and binding it to an identifier.

2.3.4 Records and procedures

We explain why we chose records and procedures as basic concepts in the kernel language. This section is intended for readers with some programming experience who wonder why we designed the kernel language the way we did.

The power of records

Records are the basic way to structure data. They are the building blocks of most data structures including lists, trees, queues, graphs, etc., as we will see in chapter 3. Records play this role to some degree in most programming languages. But we shall see that their power can go much beyond this role. The extra power appears in greater or lesser degree depending on how well or how poorly the language supports them. For maximum power, the language should make it easy to create them, take them apart, and manipulate them. In the declarative model, a record is created by simply writing it down, with a compact syntax. A record is taken apart by simply writing down a pattern, also with a compact syntax. Finally, there are many operations to manipulate records: to add, remove, or select fields; to convert to a list and back, etc. In general, languages that provide this level of support for records are called symbolic languages.

When records are strongly supported, they can be used to increase the effectiveness of many other techniques. This book focuses on three in particular: object-oriented programming, graphical user interface (GUI) design, and component-based programming. In object-oriented programming, chapter 7 shows how records can represent messages and method heads, which are what objects use to communicate. In GUI design, chapter 10 shows how records can represent "widgets," the basic building blocks of a user interface. In component-based programming, section 3.9

Operation	Description	Argument type
`A==B`	Equality comparison	Value
`A\=B`	Inequality comparison	Value
`{IsProcedure P}`	Test if procedure	Value
`A=<B`	Less than or equal comparison	Number or Atom
`A<B`	Less than comparison	Number or Atom
`A>=B`	Greater than or equal comparison	Number or Atom
`A>B`	Greater than comparison	Number or Atom
`A+B`	Addition	Number
`A-B`	Subtraction	Number
`A*B`	Multiplication	Number
`A` **`div`** `B`	Division	Int
`A` **`mod`** `B`	Modulo	Int
`A/B`	Division	Float
`{Arity R}`	Arity	Record
`{Label R}`	Label	Record
`R.F`	Field selection	Record

Table 2.3: Examples of basic operations.

then `{Arity X}=[age name]`, `{Label X}=person`, and `X.age=25`. The call to `Arity` returns a list that contains first the integer features in ascending order and then the atom features in ascending lexicographic order.

- *Comparisons.* The boolean comparison functions include `==` and `\=`, which can compare any two values for equality, as well as the numeric comparisons `=<`, `<`, `>=`, and `>`, which can compare two integers, two floats, or two atoms. Atoms are compared according to the lexicographic order of their print representations. In the following example, `Z` is bound to the maximum of `X` and `Y`:

```
declare X Y Z T in
X=5 Y=10
T=(X>=Y)
if T then Z=X else Z=Y end
```

There is syntactic sugar so that an **`if`** statement accepts an expression as its condition. The above example can be rewritten as:

```
declare X Y Z in
X=5 Y=10
if X>=Y then Z=X else Z=Y end
```

- *Procedure operations.* There are three basic operations on procedures: defining them (with the **`proc`** statement), calling them (with the curly brace notation), and testing whether a value is a procedure with the `IsProcedure` function. The call `{IsProcedure P}` returns **`true`** if `P` is a procedure and **`false`** otherwise.

```
local X in
   X=1
   local X in
      X=2
      {Browse X}
   end
   {Browse X}
end
```

What does it display? It displays first 2 and then 1. There is just one identifier, X, but at different points during the execution, it refers to different variables.

Let us summarize this idea. The meaning of an identifier like X is determined by the innermost **local** statement that declares X. The area of the program where X keeps this meaning is called the scope of X. We can find out the scope of an identifier by simply inspecting the text of the program; we do not have to do anything complicated like execute or analyze the program. This scoping rule is called lexical scoping or static scoping. Later we will see another kind of scoping rule, dynamic scoping, that is sometimes useful. But lexical scoping is by far the most important kind of scoping rule. One reason is because it is localized, i.e., the meaning of an identifier can be determined by looking at a small part of the program. We will see another reason shortly.

Procedures

Procedures are one of the most important basic building blocks of any language. We give a simple example that shows how to define and call a procedure. Here is a procedure that binds Z to the maximum of X and Y:

```
proc {Max X Y ?Z}
   if X>=Y then Z=X else Z=Y end
end
```

To make the definition easier to read, we mark the output argument with a question mark "?". This has absolutely no effect on execution; it is just a comment. Calling {Max 3 5 C} binds C to 5. How does the procedure work, exactly? When Max is called, the identifiers X, Y, and Z are bound to 3, 5, and the unbound variable referenced by C. When Max binds Z, then it binds this variable. Since C also references this variable, this also binds C. This way of passing parameters is called call by reference. Procedures output results by being passed references to unbound variables, which are bound inside the procedure. This book mostly uses call by reference, both for dataflow variables and for mutable variables. Section 6.4.4 explains some other parameter-passing mechanisms.

Procedures with external references

Let us examine the body of Max. It is just an **if** statement:

```
if X>=Y then Z=X else Z=Y end
```

This statement has one particularity, though: it cannot be executed! This is because it does not define the identifiers X, Y, and Z. These undefined identifiers are called free identifiers. (Sometimes they are called free variables, although strictly speaking they are not variables.) When put inside the procedure Max, the statement can be executed, because all the free identifiers are declared as procedure arguments.

What happens if we define a procedure that only declares some of the free identifiers as arguments? For example, let's define the procedure LB with the same procedure body as Max, but only two arguments:

```
proc {LB X ?Z}
   if X>=Y then Z=X else Z=Y end
end
```

What does this procedure do when executed? Apparently, it takes any number X and binds Z to X if X>=Y, but to Y otherwise. That is, Z is always at least Y. What is the value of Y? It is not one of the procedure arguments. It has to be the value of Y when the procedure is defined. This is a consequence of static scoping. If Y=9 when the procedure is defined, then calling {LB 3 Z} binds Z to 9. Consider the following program fragment:

```
local Y LB in
   Y=10
   proc {LB X ?Z}
      if X>=Y then Z=X else Z=Y end
   end
   local Y=15 Z in
      {LB 5 Z}
   end
end
```

What does the call {LB 5 Z} bind Z to? It will be bound to 10. The binding Y=15 when LB is called is ignored; it is the binding Y=10 at the procedure definition that is important.

Dynamic scoping versus static scoping

Consider the following simple example:

```
local P Q in
   proc {Q X} {Browse stat(X)} end
   proc {P X} {Q X} end
   local Q in
      proc {Q X} {Browse dyn(X)} end
      {P hello}
   end
end
```

What should this display, stat(hello) or dyn(hello)? Static scoping says that it will display stat(hello). In other words, P uses the version of Q that exists at P's definition. But there is another solution: P could use the version of Q that exists at P's call. This is called dynamic scoping.

Both static and dynamic scoping have been used as the default in programming languages. Let us compare the two by first giving their definitions side by side:

- *Static scope.* The variable corresponding to an identifier occurrence is the one defined in the textually innermost declaration surrounding the occurrence in the source program.
- *Dynamic scope.* The variable corresponding to an identifier occurrence is the one in the most-recent declaration seen during the execution leading up to the current statement.

The original Lisp language was dynamically scoped. Common Lisp and Scheme, which are descended from Lisp, are statically scoped by default. Common Lisp allows declaring dynamically scoped variables, which it calls special variables [200]. Which default is the right one? The correct default is procedure values with static scoping. This is because a procedure that works when it is defined will continue to work, independent of the environment where it is called. This is an important software engineering property.

Dynamic scoping remains useful in some well-defined areas. For example, consider the case of a procedure whose code is transferred across a network from one computer to another. Some of this procedure's external references, e.g., calls to common library operations, can use dynamic scoping. This way, the procedure will use local code for these operations instead of remote code. This is much more efficient.[7]

Procedural abstraction

Let us summarize what we learned from `Max` and `LB`. Three concepts play an important role:

1. Procedural abstraction. Any statement can be made into a procedure by putting it inside a procedure declaration. This is called procedural abstraction. We also say that the statement is abstracted into a procedure.

2. Free identifiers. A free identifier in a statement is an identifier that is not defined in that statement. It might be defined in an enclosing statement.

3. Static scoping. A procedure can have external references, which are free identifiers in the procedure body that are not declared as arguments. `LB` has one external reference. `Max` has none. The value of an external reference is its value when the procedure is defined. This is a consequence of static scoping.

Procedural abstraction and static scoping together form one of the most powerful tools presented in the book. In the semantics, we will see that they can be

7. However, there is no guarantee that the operation will behave in the same way on the target machine. So even for distributed programs the default should be static scoping.

implemented in a simple way.

Dataflow behavior

In the single-assignment store, variables can be unbound. On the other hand, some statements need bound variables, otherwise they cannot execute. For example, what happens when we execute:

```
local X Y Z in
   X=10
   if X>=Y then Z=X else Z=Y end
end
```

The comparison `X>=Y` returns **`true`** or **`false`**, if it can decide which is the case. If `Y` is unbound, it cannot decide, strictly speaking. What does it do? Continuing with either **`true`** or **`false`** would be incorrect. Raising an error would be a drastic measure, since the program has done nothing wrong (it has done nothing right either). We decide that the program will simply stop its execution, without signaling any kind of error. If some other activity (to be determined later) binds `Y`, then the stopped execution can continue as if nothing had perturbed the normal flow of execution. This is called dataflow behavior. Dataflow behavior underlies a second powerful tool presented in the book, namely concurrency. In the semantics, we will see that dataflow behavior can be implemented in a simple way.

2.4.2 The abstract machine

We define the semantics of the kernel language as an operational semantics, i.e., it defines the meaning of the kernel language through its execution on an abstract machine. We first define the basic concepts of the abstract machine: environments, semantic statement, statement stack, execution state, and computation. We then show how to execute a program. Finally, we explain how to calculate with environments, which is a common semantic operation.

Definitions

A running program is defined in terms of a computation, which is a sequence of execution states. Let us define exactly what this means. We need the following concepts:

- A *single-assignment store* σ is a set of store variables. These variables are partitioned into (1) sets of variables that are equal but unbound and (2) variables that are bound to a number, record, or procedure. For example, in the store $\{x_1, x_2 = x_3, x_4 = \mathtt{a}\,|\,x_2\}$, x_1 is unbound, x_2 and x_3 are equal and unbound, and x_4 is bound to the partial value $\mathtt{a}\,|\,x_2$. A store variable bound to a value is indistinguishable from that value. This is why a store variable is sometimes called a store entity.

- At each step, the first element of ST is popped and execution proceeds according to the form of the element.
- The final execution state (if there is one) is a state in which the semantic stack is empty.

A semantic stack ST can be in one of three run-time states:

- Runnable: ST can do a computation step.
- Terminated: ST is empty.
- Suspended: ST is not empty, but it cannot do any computation step.

Calculating with environments

A program execution often does calculations with environments. An environment E is a function that maps variable identifiers $\langle \mathsf{x} \rangle$ to store entities (both unbound variables and values). The notation $E(\langle \mathsf{x} \rangle)$ retrieves the entity associated with the identifier $\langle \mathsf{x} \rangle$ from the store. To define the semantics of the abstract machine instructions, we need two common operations on environments, namely adjunction and restriction.

Adjunction defines a new environment by adding a mapping to an existing one. The notation

$$E + \{\langle \mathsf{x} \rangle \rightarrow x\}$$

denotes a new environment E' constructed from E by adding the mapping $\{\langle \mathsf{x} \rangle \rightarrow x\}$. This mapping overrides any other mapping from the identifier $\langle \mathsf{x} \rangle$. That is, $E'(\langle \mathsf{x} \rangle)$ is equal to x, and $E'(\langle \mathsf{y} \rangle)$ is equal to $E(\langle \mathsf{y} \rangle)$ for all identifiers $\langle \mathsf{y} \rangle$ different from $\langle \mathsf{x} \rangle$. When we need to add more than one mapping at once, we write $E + \{\langle \mathsf{x} \rangle_1 \rightarrow x_1, \ldots, \langle \mathsf{x} \rangle_n \rightarrow x_n\}$.

Restriction defines a new environment whose domain is a subset of an existing one. The notation

$$E|_{\{\langle \mathsf{x} \rangle_1, \ldots, \langle \mathsf{x} \rangle_n\}}$$

denotes a new environment E' such that $\mathrm{dom}(E') = \mathrm{dom}(E) \cap \{\langle \mathsf{x} \rangle_1, \ldots, \langle \mathsf{x} \rangle_n\}$ and $E'(\langle \mathsf{x} \rangle) = E(\langle \mathsf{x} \rangle)$ for all $\langle \mathsf{x} \rangle \in \mathrm{dom}(E')$. That is, the new environment does not contain any identifiers other than those mentioned in the set.

2.4.3 Nonsuspendable statements

We first give the semantics of the statements that can never suspend.

The `skip` statement

The semantic statement is:

identifiers in ⟨v⟩ are replaced by their store contents as given by E.

- Bind $E(\langle x \rangle)$ and x in the store.

We have seen how to construct record and number values, but what about procedure values? In order to explain them, we have first to explain the concept of lexical scoping.

Free and bound identifier occurrences

A statement ⟨s⟩ can contain many occurrences of variable identifiers. For each identifier occurrence, we can ask the question: where was this identifier declared? If the declaration is in some statement (part of ⟨s⟩ or not) that textually surrounds (i.e., encloses) the occurrence, then we say that the declaration obeys lexical scoping. Because the scope is determined by the source code text, this is also called static scoping.

Identifier occurrences in a statement can be bound or free with respect to that statement. An identifier occurrence X is bound with respect to a statement ⟨s⟩ if it is declared inside ⟨s⟩, i.e., in a **local** statement, in the pattern of a **case** statement, or as argument of a procedure declaration. An identifier occurrence that is not bound is free. Free occurrences can only exist in incomplete program fragments, i.e., statements that cannot run. In a running program, it is always true that every identifier occurrence is bound.

> **Bound identifier occurrences and bound variables**
>
> Do not confuse a bound identifier occurrence with a bound variable! A bound identifier occurrence does not exist at run time; it is a textual variable name that textually occurs inside a construct that declares it (e.g., a procedure or variable declaration). A bound variable exists at run time; it is a dataflow variable that is bound to a partial value.

Here is an example with both free and bound occurrences:

```
local Arg1 Arg2 in
   Arg1=111*111
   Arg2=999*999
   Res=Arg1+Arg2
end
```

In this statement, all variable identifiers are declared with lexical scoping. All occurrences of the identifiers `Arg1` and `Arg2` are bound and the single occurrence of `Res` is free. This statement cannot be run. To make it runnable, it has to be part of a bigger statement that declares `Res`. Here is an extension that can run:

```
local Res in
   local Arg1 Arg2 in
      Arg1=111*111
      Arg2=999*999
      Res=Arg1+Arg2
   end
   {Browse Res}
end
```

This can run since it has no free identifier occurrences.

Procedure values (closures)

Let us see how to construct a procedure value in the store. It is not as simple as one might imagine because procedures can have external references. For example:

```
proc {LowerBound X ?Z}
   if X>=Y then Z=X else Z=Y end
end
```

In this example, the **if** statement has three free variables, X, Y, and Z. Two of them, X and Z, are also formal parameters. The third, Y, is not a formal parameter. It has to be defined by the environment where the procedure is declared. The procedure value itself must have a mapping from Y to the store. Otherwise, we could not call the procedure since Y would be a kind of dangling reference.

Let us see what happens in the general case. A procedure expression is written as:

proc { $ ⟨y⟩$_1$ ⋯ ⟨y⟩$_n$ } ⟨s⟩ **end**

The statement ⟨s⟩ can have free variable identifiers. Each free identifier is either a formal parameter or not. The first kind are defined anew each time the procedure is called. They form a subset of the formal parameters $\{\langle y\rangle_1, \ldots, \langle y\rangle_n\}$. The second kind are defined once and for all when the procedure is declared. We call them the external references of the procedure. Let us write them as $\{\langle z\rangle_1, \ldots, \langle z\rangle_k\}$. Then the procedure value is a pair:

(**proc** { $ ⟨y⟩$_1$ ⋯ ⟨y⟩$_n$ } ⟨s⟩ **end**, CE)

Here CE (the contextual environment) is $E|_{\{\langle z\rangle_1,\ldots,\langle z\rangle_n\}}$, where E is the environment when the procedure is declared. This pair is put in the store just like any other value.

Because it contains an environment as well as a procedure definition, a procedure value is often called a closure or a lexically scoped closure. This is because it "closes" (i.e., packages up) the environment at procedure definition time. This is also called environment capture. When the procedure is called, the contextual environment is used to construct the environment of the executing procedure body.

2.4.4 Suspendable statements

There are three statements remaining in the kernel language:

⟨s⟩ ::= ...
| **if** ⟨x⟩ **then** ⟨s⟩$_1$ **else** ⟨s⟩$_2$ **end**
| **case** ⟨x⟩ **of** ⟨pattern⟩ **then** ⟨s⟩$_1$ **else** ⟨s⟩$_2$ **end**
| ´{´ ⟨x⟩ ⟨y⟩$_1$ ··· ⟨y⟩$_n$ ´}´

What should happen with these statements if ⟨x⟩ is unbound? From the discussion in section 2.2.8, we know what should happen. The statements should simply wait until ⟨x⟩ is bound. We say that they are suspendable statements. They have an *activation condition*, which we define as a condition that must be true for execution to continue. The condition is that $E(\langle x \rangle)$ must be determined, i.e., bound to a number, record, or procedure.

In the declarative model of this chapter, once a statement suspends it will never continue. The program simply stops executing. This is because there is no other execution that could make the activation condition true. In chapter 4, when we introduce concurrent programming, we will have executions with more than one semantic stack. A suspended stack ST can become runnable again if another stack does an operation that makes ST's activation condition true. This is the basis of dataflow execution. For now, let us investigate sequential programming and stick with just a single semantic stack.

Conditional (the `if` *statement)*

The semantic statement is:

(**if** ⟨x⟩ **then** ⟨s⟩$_1$ **else** ⟨s⟩$_2$ **end**, E)

Execution consists of the following actions:

- If the activation condition is true ($E(\langle x \rangle)$ is determined), then do the following actions:
  - If $E(\langle x \rangle)$ is not a boolean (**true** or **false**) then raise an error condition.
  - If $E(\langle x \rangle)$ is **true**, then push $(\langle s \rangle_1, E)$ on the stack.
  - If $E(\langle x \rangle)$ is **false**, then push $(\langle s \rangle_2, E)$ on the stack.
- If the activation condition is false, then execution does not continue. The execution state is kept as is. We say that execution suspends. The stop can be temporary. If some other activity in the system makes the activation condition true, then execution can resume.

Procedure application

The semantic statement is:

$(\{\langle x \rangle\ \langle y \rangle_1 \cdots \langle y \rangle_n\}, E)$

Execution consists of the following actions:

- If the activation condition is true ($E(\langle x \rangle)$ is determined), then do the following actions:
  - If $E(\langle x \rangle)$ is not a procedure value or is a procedure with a number of arguments different from n, then raise an error condition.
  - If $E(\langle x \rangle)$ has the form $(\texttt{proc}\ \{\ \$\ \langle z \rangle_1 \cdots \langle z \rangle_n\}\ \langle s \rangle\ \texttt{end}, CE)$ then push $(\langle s \rangle, CE + \{\langle z \rangle_1 \rightarrow E(\langle y \rangle_1), \ldots, \langle z \rangle_n \rightarrow E(\langle y \rangle_n)\})$ on the stack.
- If the activation condition is false, then suspend execution.

Pattern matching (the `case` *statement)*

The semantic statement is:

$(\texttt{case}\ \langle x \rangle\ \texttt{of}\ \langle lit \rangle(\langle feat \rangle_1 \colon \langle x \rangle_1 \cdots \langle feat \rangle_n \colon \langle x \rangle_n)\ \texttt{then}\ \langle s \rangle_1\ \texttt{else}\ \langle s \rangle_2\ \texttt{end}, E)$

(Here ⟨lit⟩ and ⟨feat⟩ are synonyms for ⟨literal⟩ and ⟨feature⟩.) Execution consists of the following actions:

- If the activation condition is true ($E(\langle x \rangle)$ is determined), then do the following actions:
  - If the label of $E(\langle x \rangle)$ is ⟨lit⟩ and its arity is $[\langle feat \rangle_1 \cdots \langle feat \rangle_n]$, then push $(\langle s \rangle_1, E + \{\langle x \rangle_1 \rightarrow E(\langle x \rangle).\langle feat \rangle_1, \ldots, \langle x \rangle_n \rightarrow E(\langle x \rangle).\langle feat \rangle_n\})$ on the stack.
  - Otherwise push $(\langle s \rangle_2, E)$ on the stack.
- If the activation condition is false, then suspend execution.

2.4.5 Basic concepts revisited

Now that we have seen the kernel semantics, let us look again at the examples of section 2.4.1 to see exactly what they are doing. We look at three examples; we suggest you do the others as exercises.

Variable identifiers and static scoping

We saw before that the following statement ⟨s⟩ displays first 2 and then 1:

$$\langle s \rangle \equiv \left\{ \begin{array}{l} \texttt{local X in} \\ \quad \texttt{X=1} \\ \quad \langle s \rangle_1 \equiv \left\{ \begin{array}{l} \texttt{local X in} \\ \quad \texttt{X=2} \\ \quad \texttt{\{Browse X\}} \\ \texttt{end} \end{array} \right. \\ \quad \langle s \rangle_2 \equiv \texttt{\{Browse X\}} \\ \texttt{end} \end{array} \right.$$

```
local Max C in
   proc {Max X Y ?Z}
      if X>=Y then Z=X else Z=Y end
   end
   {Max 3 5 C}
end
```

Translating into kernel syntax and rearranging slightly gives

$\langle s \rangle \equiv$

```
local Max in
   local A in
      local B in
         local C in
            Max=proc {$ X Y Z}
                   local T in
                      T=(X>=Y)
                      if T then Z=X else Z=Y end
                   end
                end
            A=3
            B=5
            {Max A B C}
         end
      end
   end
end
```

You can see that the kernel syntax is rather verbose. This is because of the simplicity of the kernel language. This simplicity is important because it lets us keep the semantics simple too. The original source code uses the following three syntactic shortcuts for readability:

- Declaring more than one variable in a **local** declaration. This is translated into nested **local** declarations.
- Using "in-line" values instead of variables, e.g., `{P 3}` is a shortcut for **local** `X` **in** `X=3 {P X}` **end**.
- Using nested operations, e.g., putting the operation `X>=Y` in place of the boolean in the **if** statement.

We will use these shortcuts in all examples from now on.

Let us now execute statement $\langle s \rangle$. For clarity, we omit some of the intermediate steps.

```
local LowerBound Y C in
   Y=5
   proc {LowerBound X ?Z}
      if X>=Y then Z=X else Z=Y end
   end
   {LowerBound 3 C}
end
```

This is very close to the `Max` example. The body of `LowerBound` is identical to the body of `Max`. The only difference is that `LowerBound` has an external reference. The procedure value is

(**proc** {$ X Z} **if** X>=Y **then** Z=X **else** Z=Y **end end**, $\{\texttt{Y} \rightarrow y\}$)

where the store contains:

$y = 5$

When the procedure is defined, i.e., when the procedure value is created, the environment has to contain a mapping of `Y`. Now let us apply this procedure. We assume that the procedure is called as {`LowerBound A C`}, where `A` is bound to 3. Before the application we have:

($[(\{\texttt{LowerBound A C}\}, \{\texttt{Y} \rightarrow y, \texttt{LowerBound} \rightarrow lb, \texttt{A} \rightarrow a, \texttt{C} \rightarrow c\})]$,
{ $lb = ($**proc** {$ X Z} **if** X>=Y **then** Z=X **else** Z=Y **end end**$, \{\texttt{Y} \rightarrow y\})$,
$y = 5, a = 3, c\}$)

After the application we get:

($[($**if** X>=Y **then** Z=X **else** Z=Y **end**$, \{\texttt{Y} \rightarrow y, \texttt{X} \rightarrow a, \texttt{Z} \rightarrow c\})]$,
{ $lb = ($**proc** {$ X Z} **if** X>=Y **then** Z=X **else** Z=Y **end end**$, \{\texttt{Y} \rightarrow y\})$,
$y = 5, a = 3, c\}$)

The new environment is calculated by starting with the contextual environment ($\{\texttt{Y} \rightarrow y\}$ in the procedure value) and adding mappings from the formal arguments `X` and `Z` to the actual arguments a and c.

Procedure with external references (part 2)

In the above execution, the identifier `Y` refers to y in both the calling environment as well as the contextual environment of `LowerBound`. How would the execution change if the following statement were executed instead of {`LowerBound 3 C`}?:

```
local Y in
   Y=10
   {LowerBound 3 C}
end
```

Here `Y` no longer refers to y in the calling environment. Before looking at the answer, please put down the book, take a piece of paper, and work it out. Just before the application we have almost the same situation as before:

($[(\{\texttt{LowerBound A C}\}, \{\texttt{Y} \rightarrow y', \texttt{LowerBound} \rightarrow lb, \texttt{A} \rightarrow a, \texttt{C} \rightarrow c\})]$,
$\{\ lb = (\textbf{proc}\ \{\$\ \texttt{X Z}\}\ \textbf{if}\ \texttt{X>=Y}\ \textbf{then}\ \texttt{Z=X}\ \textbf{else}\ \texttt{Z=Y}\ \textbf{end end}, \{\texttt{Y} \rightarrow y\}),$
$y' = 10, y = 5, a = 3, c\}$)

The calling environment has changed slightly: `Y` refers to a new variable y', which is bound to 10. When doing the application, the new environment is calculated in exactly the same way as before, starting from the contextual environment and adding the formal arguments. This means that the y' is ignored! We get exactly the same situation as before in the semantic stack:

($[(\textbf{if}\ \texttt{X>=Y}\ \textbf{then}\ \texttt{Z=X}\ \textbf{else}\ \texttt{Z=Y}\ \textbf{end}, \{\texttt{Y} \rightarrow y, \texttt{X} \rightarrow a, \texttt{Z} \rightarrow c\})]$,
$\{\ lb = (\textbf{proc}\ \{\$\ \texttt{X Z}\}\ \textbf{if}\ \texttt{X>=Y}\ \textbf{then}\ \texttt{Z=X}\ \textbf{else}\ \texttt{Z=Y}\ \textbf{end end}, \{\texttt{Y} \rightarrow y\}),$
$y' = 10, y = 5, a = 3, c\}$)

The store still has the binding $y' = 10$. But y' is not referenced by the semantic stack, so this binding makes no difference to the execution.

2.5 Memory management

The abstract machine we defined in the previous section is a powerful tool with which we can investigate properties of computations. As a first exploration, let us look at memory behavior, i.e., how the sizes of the semantic stack and store evolve as a computation progresses. We will look at the principle of last call optimization and explain it by means of the abstract machine. This will lead us to the concepts of memory life cycle and garbage collection.

2.5.1 Last call optimization

Consider a recursive procedure with just one recursive call which happens to be the last call in the procedure body. We call such a procedure tail-recursive. We will show that the abstract machine executes a tail-recursive procedure with a constant stack size. This property is called last call optimization or tail call optimization. The term tail recursion optimization is sometimes used, but is less precise since the optimization works for any last call, not just tail-recursive calls (see Exercises, section 2.9). Consider the following procedure:

```
proc {Loop10 I}
   if I==10 then skip
   else
      {Browse I}
      {Loop10 I+1}
   end
end
```

Calling `{Loop10 0}` displays successive integers from 0 up to 9. Let us see how this procedure executes.

- The initial execution state is:

$$\begin{array}{l}(\ [(\{\texttt{Loop10 0}\}, E_0)], \\ \ \ \sigma \ )\end{array}$$

where E_0 is the environment at the call and σ the initial store.

- After executing the `if` statement, this becomes:

$$\begin{array}{l}(\ [(\{\texttt{Browse I}\}, \{\texttt{I} \rightarrow i_0\}) \ \ (\{\texttt{Loop10 I+1}\}, \{\texttt{I} \rightarrow i_0\})], \\ \ \ \{i_0 = 0\} \cup \sigma \ )\end{array}$$

- After executing the `Browse`, we get to the first recursive call:

$$\begin{array}{l}(\ [(\{\texttt{Loop10 I+1}\}, \{\texttt{I} \rightarrow i_0\})], \\ \ \ \{i_0 = 0\} \cup \sigma \ )\end{array}$$

- After executing the `if` statement in the recursive call, this becomes:

$$\begin{array}{l}(\ [(\{\texttt{Browse I}\}, \{\texttt{I} \rightarrow i_1\}) \ \ (\{\texttt{Loop10 I+1}\}, \{\texttt{I} \rightarrow i_1\})], \\ \ \ \{i_0 = 0, i_1 = 1\} \cup \sigma \ )\end{array}$$

- After executing the `Browse` again, we get to the second recursive call:

$$\begin{array}{l}(\ [(\{\texttt{Loop10 I+1}\}, \{\texttt{I} \rightarrow i_1\})], \\ \ \ \{i_0 = 0, i_1 = 1\} \cup \sigma \ )\end{array}$$

It is clear that the stack at the kth recursive call is always of the form:

$$[(\{\texttt{Loop10 I+1}\}, \{\texttt{I} \rightarrow i_{k-1}\})]$$

There is just one semantic statement and its environment is of constant size. This is the last call optimization. This shows that the efficient way to program loops in the declarative model is to program the loop as a tail-recursive procedure.

We can further see that the sizes of the semantic stack and the store evolve quite differently. The semantic stack is bounded by a constant size. On the other hand, the store grows bigger at each call. At the kth recursive call, the store has the form:

$$\{i_0 = 0, i_1 = 1, \ldots, i_{k-1} = k-1\} \cup \sigma$$

The store size is proportional to the number of recursive calls. Let us see why this growth is not a problem in practice. Look carefully at the semantic stack of the kth recursive call. It does not need the variables $\{i_0, i_1, \ldots, i_{k-2}\}$. The only variable needed is i_{k-1}. This means that we can remove the not-needed variables from the store without changing the results of the computation. This gives a smaller store:

$$\{i_{k-1} = k-1\} \cup \sigma$$

This smaller store is of constant size. If we could somehow ensure that the not-needed variables are always removed, then the program could execute indefinitely with a constant memory size.

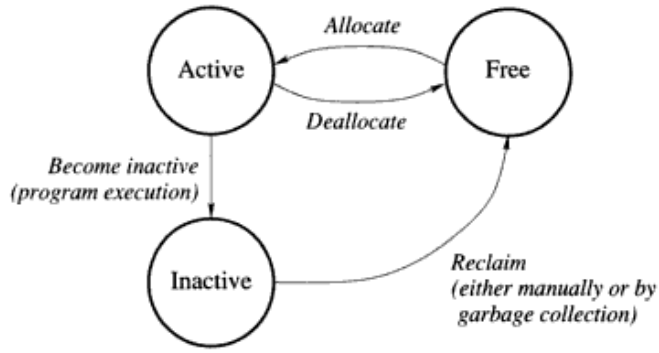


Figure 2.18: Life cycle of a memory block.

For this example we can solve the problem by storing the variables on the stack instead of in the store. This is possible because the variables are bound to small integers that fit in a machine word (e.g., 32 bits). So the stack can store the integers directly instead of storing variable references into the store.[8] This example is atypical; almost all realistic programs have large amounts of long-lived or shared data that cannot easily be stored on the stack. So we still have to solve the general problem of removing the not-needed variables. In the next sections we will see how to do this.

2.5.2 Memory life cycle

From the abstract machine semantics it follows that a running program needs only the information in the semantic stack and in that part of the store reachable from the semantic stack. A partial value is reachable if it is referenced by a statement on the semantic stack or by another reachable partial value. The semantic stack and the reachable part of the store are together called the active memory. All the memory that is not active can safely be reclaimed, i.e., it can be reused in the computation. We saw that the active memory size of the `Loop10` example is bounded by a constant, which means that it can loop indefinitely without exhausting system memory.

Now we can introduce the concept of memory life cycle. Programs execute in main memory, which consists of a sequence of memory words. At the time of writing, low-cost personal computers have 32-bit words and high-end computers have 64-bit or longer words. The sequence of words is divided up into blocks, where a block consists of a sequence of one or more words used to store part of an execution state.

8. A further optimization that is often done is to store part of the stack in machine registers. This is important since machine registers are much faster to read and write than main memory.

independently of the running program. This completely eliminates dangling references and greatly reduces memory leaks. This relieves the programmer of most of the difficulties of manual memory management. Automatic reclaiming is called garbage collection. Garbage collection is a well-known technique that has been used for a long time. It was used in the 1960s for early Lisp systems. Until the 1990s, mainstream languages did not use it because it was judged (erroneously) as being too inefficient. It has finally become acceptable in mainstream programming because of the popularity of the Java language.

A typical garbage collector has two phases. In the first phase, it determines what the active memory is. It does this by finding all data structures reachable from an initial set of pointers called the root set. The original meaning of "pointer" is an address in the address space of a process. In the context of our abstract machine, a pointer is a variable reference in the store. The root set is the set of pointers that are always needed by the program. In the abstract machine of the previous section, the root set is simply the semantic stack. In general, the root set includes all pointers in ready threads and all pointers in operating system data structures. We will see this when we extend the abstract machine in later chapters to implement new concepts. The root set also includes some pointers related to distributed programming (namely references from remote sites; see chapter 11).

In the second phase, the garbage collector compacts the memory. That is, it collects all the active memory blocks into one contiguous block (a block without holes) and the free memory blocks into one contiguous block.

Modern garbage collection algorithms are efficient enough that most applications can use them with only small memory and time penalties [107]. The most widely used garbage collectors run in a "batch" mode, i.e., they are dormant most of the time and run only when the total amount of active and inactive memory reaches a predefined threshold. While the garbage collector runs, there is a pause in program execution. Usually the pause is small enough not to be disruptive.

There exist garbage collection algorithms, called real-time garbage collectors, that can run continuously, interleaved with the program execution. They can be used in cases, such as hard real-time programming, in which there must not be any pauses.

2.5.4 Garbage collection is not magic

Having garbage collection lightens the burden of memory management for the developer, but it does not eliminate it completely. There are two cases that remain the developer's responsibility: avoiding memory leaks and managing external resources.

Avoiding memory leaks

The programmer still has some responsibility regarding memory leaks. If the program continues to reference a data structure that it no longer needs, then that

called finalization, which defines actions to be taken when data structures become unreachable. Finalization is explained in section 6.9.2.

The second case is when an external resource needs a Mozart data structure. This is often straightforward to handle. For example, consider a scenario where the Mozart program implements a database server that is accessed by external clients. This scenario has a simple solution: never do automatic reclaiming of the database storage. Other scenarios may not be so simple. A general solution is to set aside a part of the Mozart program to represent the external resource. This part should be active (i.e., have its own thread) so that it is not reclaimed haphazardly. It can be seen as a "proxy" for the resource. The proxy keeps a reference to the Mozart data structure as long as the resource needs it. The resource informs the proxy when it no longer needs the data structure. Section 6.9.2 gives another technique.

2.5.5 The Mozart garbage collector

The Mozart system does automatic memory management. It has both a local garbage collector and a distributed garbage collector. The latter is used for distributed programming and is explained in chapter 11. The local garbage collector uses a copying dual-space algorithm.

The garbage collector divides memory into two spaces, of which each takes up half of available memory space. At any instant, the running program sits completely in one half. Garbage collection is done when there is no more free memory in that half. The garbage collector finds all data structures that are reachable from the root set and copies them to the other half of memory. Since they are copied to one contiguous memory block this also does compaction.

The advantage of a copying garbage collector is that its execution time is proportional to the active memory size, not to the total memory size. Small programs will garbage-collect quickly, even if they are running in a large memory space. The two disadvantages of a copying garbage collector are that half the memory is unusable at any given time and that long-lived data structures (like system tables) have to be copied at each garbage collection. Let us see how to remove these two disadvantages. Copying long-lived data can be avoided by using a modified algorithm called a generational garbage collector. This partitions active memory into generations. Long-lived data structures are put in older generations, which are collected less often.

The memory disadvantage is only important if the active memory size approaches the maximum addressable memory size of the underlying architecture. Mainstream computer technology is currently in a transition period from 32-bit to 64-bit addressing. In a computer with 32-bit addresses, the limit is reached when active memory size is 1000 MB or more. (The limit is usually not 2^{32} bytes, i.e., 4096 MB, due to limitations in the operating system.) At the time of writing, this limit is reached by large programs in high-end personal computers. For such programs, we recommend using a computer with 64-bit addresses, which has no such problem.

2.6 From kernel language to practical language

The kernel language has all the concepts needed for declarative programming. But trying to use it for practical declarative programming shows that it is too minimal. Kernel programs are just too verbose. Most of this verbosity can be eliminated by judiciously adding syntactic sugar and linguistic abstractions. This section does just that:

- It defines a set of syntactic conveniences that give a more concise and readable full syntax.
- It defines an important linguistic abstraction, namely functions, that is useful for concise and readable programming.
- It explains the interactive interface of the Mozart system and shows how it relates to the declarative model. This brings in the **`declare`** statement, which is a variant of the **`local`** statement designed for interactive use.

The resulting language is used in chapter 3 to explain the programming techniques of the declarative model.

2.6.1 Syntactic conveniences

The kernel language defines a simple syntax for all its constructs and types. The full language has the following conveniences to make this syntax more usable:

- Nested partial values can be written in a concise way.
- Variables can be both declared and initialized in one step.
- Expressions can be written in a concise way.
- The **`if`** and **`case`** statements can be nested in a concise way.
- The operators **`andthen`** and **`orelse`** are defined for nested **`if`** statements.
- Statements can be converted into expressions by using a nesting marker.

The nonterminal symbols used in the kernel syntax and semantics correspond as follows to those in the full syntax:

Kernel syntax	Full syntax
⟨x⟩, ⟨y⟩, ⟨z⟩	⟨variable⟩
⟨s⟩	⟨statement⟩, ⟨stmt⟩

Nested partial values

In table 2.2, the syntax of records and patterns implies that their arguments are variables. In practice, many partial values are nested deeper than this. Because nested values are so often used, we give syntactic sugar for them. For example, we extend the syntax to let us write `person(name:"George" age:25)` instead of the

more cumbersome version:

```
local A B in A="George" B=25 X=person(name:A age:B) end
```

where X is bound to the nested record.

Implicit variable initialization

To make programs shorter and easier to read, there is syntactic sugar to bind a variable immediately when it is declared. The idea is to put a bind operation between **local** and **in**. Instead of **local** X **in** X=10 {Browse X} **end**, in which X is mentioned three times, the shortcut lets one write **local** X=10 **in** {Browse X} **end**, which mentions X only twice. A simple case is the following:

local X=⟨expression⟩ **in** ⟨statement⟩ **end**

This declares X and binds it to the result of ⟨expression⟩. The general case is:

local ⟨pattern⟩=⟨expression⟩ **in** ⟨statement⟩ **end**

where ⟨pattern⟩ is any partial value. This first declares all the variables in ⟨pattern⟩ and then binds ⟨pattern⟩ to the result of ⟨expression⟩. The general rule in both examples is that variable identifiers occurring on the left-hand side of the equality, i.e., X or the identifiers in ⟨pattern⟩, are the ones declared. Variable identifiers on the right-hand side are not declared.

Implicit variable initialization is convenient for building a complex data structure when we need variable references inside the structure. For example, if T is unbound, then the following:

local tree(key:A left:B right:C value:D)=T **in** ⟨statement⟩ **end**

builds the tree record, binds it to T, and declares A, B, C, and D as referring to parts of T. This is strictly equivalent to:

```
local A B C D in
   T=tree(key:A left:B right:C value:D) ⟨statement⟩
end
```

It is interesting to compare implicit variable initialization with the **case** statement. Both use patterns and implicitly declare variables. The first uses them to build data structures and the second uses them to take data structures apart.[10]

10. Implicit variable initialization can also be used to take data structures apart. If T is already bound to a tree record, then its four fields will be bound to A, B, C, and D. This works because the binding operation is actually doing unification, which is a symmetric operation (see section 2.8.2). We do not recommend this use.

```
⟨expression⟩ ::= ⟨variable⟩ | ⟨int⟩ | ⟨float⟩
               | ⟨unaryOp⟩ ⟨expression⟩
               | ⟨expression⟩ ⟨evalBinOp⟩ ⟨expression⟩
               | "(" ⟨expression⟩ ")"
               | "{" ⟨expression⟩ { ⟨expression⟩ } "}"
               | ...
⟨unaryOp⟩    ::= "~" | ...
⟨evalBinOp⟩  ::= "+" | "-" | "*" | "/" | div | mod
               | "==" | "\=" | "<" | "=<" | ">" | ">=" | ...
```

Table 2.4: Expressions for calculating with numbers.

```
⟨statement⟩    ::= if ⟨expression⟩ then ⟨inStatement⟩
                   { elseif ⟨expression⟩ then ⟨inStatement⟩ }
                   [ else ⟨inStatement⟩ ] end
                 | ...
⟨inStatement⟩  ::= [ { ⟨declarationPart⟩ }+ in ] ⟨statement⟩
```

Table 2.5: The `if` statement.

Expressions

An expression is syntactic sugar for a sequence of operations that returns a value. It is different from a statement, which is also a sequence of operations but does not return a value. An expression can be used inside a statement whenever a value is needed. For example, `11*11` is an expression and `X=11*11` is a statement. Semantically, an expression is defined by a straightforward translation into kernel syntax. So `X=11*11` is translated into `{Mul 11 11 X}`, where `Mul` is a three-argument procedure that does multiplication.[11]

Table 2.4 shows the syntax of expressions that calculate with numbers. Later on we will see expressions for calculating with other data types. Expressions are built hierarchically, starting from basic expressions (e.g., variables and numbers) and combining them together. There are two ways to combine them: using operators (e.g., the addition `1+2+3+4`) or using function calls (e.g., the square root `{Sqrt 5.0}`).

11. Its real name is `Number."*"`, since it is part of the `Number` module.

clause only if the clause's pattern is inconsistent with the input argument.

Nested patterns are handled by looking first at the outermost pattern and then working inward. The nested pattern `(X|Xr)#(Y|Yr)` has one outer pattern of the form `A#B` and two inner patterns of the form `A|B`. All three patterns are tuples that are written with infix syntax, using the infix operators `'#'` and `'|'`. They could have been written with the usual syntax as `'#'(A B)` and `'|'(A B)`. Each inner pattern `(X|Xr)` and `(Y|Yr)` is put in its own primitive **case** statement. The outer pattern using `'#'` disappears from the translation because it occurs also in the **case**'s input argument. The matching with `'#'` can therefore be done at translation time.

The operators **andthen** *and* **orelse**

The operators **andthen** and **orelse** are used in calculations with boolean values. The expression

⟨expression⟩$_1$ **andthen** ⟨expression⟩$_2$

translates into

if ⟨expression⟩$_1$ **then** ⟨expression⟩$_2$ **else false end**

The advantage of using **andthen** is that ⟨expression⟩$_2$ is not evaluated if ⟨expression⟩$_1$ is **false**. There is an analogous operator **orelse**. The expression

⟨expression⟩$_1$ **orelse** ⟨expression⟩$_2$

translates into

if ⟨expression⟩$_1$ **then true else** ⟨expression⟩$_2$ **end**

That is, ⟨expression⟩$_2$ is not evaluated if ⟨expression⟩$_1$ is **true**.

Nesting markers

The nesting marker "`$`" turns any statement into an expression. The expression's value is what is at the position indicated by the nesting marker. For example, the statement `{P X1 X2 X3}` can be written as `{P X1 $ X3}`, which is an expression whose value is `X2`. This makes the source code more concise, since it avoids having to declare and use the identifier `X2`. The variable corresponding to `X2` is hidden from the source code.

Nesting markers can make source code more readable to a proficient programmer, while making it harder for a beginner to see how the code translates to the kernel language. We use them only when they greatly increase readability. For example, instead of writing

```
local X in {Obj get(X)} {Browse X} end
```

The extra argument R is bound to the expression in the procedure body. If the function body is an **if** statement, then each alternative of the **if** can end in an expression:

```
fun {Max X Y}
   if X>=Y then X else Y end
end
```

This translates to:

```
proc {Max X Y ?R}
   R = if X>=Y then X else Y end
end
```

We can further translate this by transforming the **if** from an expression to a statement. This gives the final result:

```
proc {Max X Y ?R}
   if X>=Y then R=X else R=Y end
end
```

Similar rules apply for the **local** and **case** statements, and for other statements we will see later. Each statement can be used as an expression. Roughly speaking, whenever an execution sequence in a procedure ends in a statement, the corresponding sequence in a function ends in an expression. Table 2.7 gives the complete syntax of expressions after applying this rule. This table takes all the statements we have seen so far and shows how to use them as expressions. In particular, there are also function values, which are simply procedure values written in functional syntax.

Function calls

A function call {F X1 ... XN} translates to the procedure call {F X1 ... XN R}, where R replaces the function call where it is used. For example, the following nested call of F:

```
{Q {F X1 ... XN} ...}
```

is translated to:

```
local R in
   {F X1 ... XN R}
   {Q R ...}
end
```

In general, nested function calls are evaluated before the function call in which they are nested. If there are several, then they are evaluated in the order they appear in the program.

Function calls in data structures

There is one more rule to remember for function calls. It has to do with a call inside a data structure (record, tuple, or list). Here is an example:

⟨interStatement⟩ ::= ⟨statement⟩ | **declare** { ⟨declarationPart⟩ }+ [⟨interStatement⟩] | **declare** { ⟨declarationPart⟩ }+ **in** ⟨interStatement⟩ ⟨declarationPart⟩ ::= ⟨variable⟩ | ⟨pattern⟩ "=" ⟨expression⟩ | ⟨statement⟩

Table 2.8: Interactive statement syntax.

```
Ys={F X}|{Map Xr F}
```

In this case, the translation puts the nested calls after the bind operation:

```
local Y Yr in
   Ys=Y|Yr
   {F X Y}
   {Map Xr F Yr}
end
```

This ensures that the recursive call is last. Section 2.5.1 explains why this is important for execution efficiency. The full `Map` function is defined as follows:

```
fun {Map Xs F}
   case Xs
   of nil then nil
   [] X|Xr then {F X}|{Map Xr F}
   end
end
```

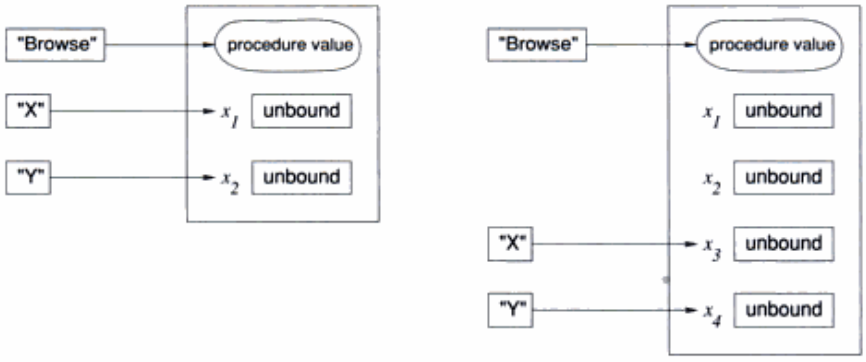
`Map` applies the function `F` to all elements of a list and returns the result. Here is an example call:

```
{Browse {Map [1 2 3 4] fun {$ X} X*X end}}
```

This displays `[1 4 9 16]`. The definition of `Map` translates as follows to the kernel language:

```
proc {Map Xs F ?Ys}
   case Xs of nil then Ys=nil
   else case Xs of X|Xr then
      local Y Yr in
         Ys=Y|Yr {F X Y} {Map Xr F Yr}
      end
   end end
end
```

The dataflow variable `Yr` is used as a "placeholder" for the result in the recursive call `{Map Xr F Yr}`. This lets the recursive call be the last call. In our model, this means that the recursion executes with the same space and time efficiency as an iterative construct like a **while** loop.



Result of first `declare X Y` Result of second `declare X Y`

Figure 2.19: Declaring global variables.

2.6.3 Interactive interface (the `declare` statement)

The Mozart system has an interactive interface that allows introducing program fragments incrementally and execute them as they are introduced. The fragments have to respect the syntax of interactive statements, which is given in table 2.8. An interactive statement is either any legal statement or a new form, the **`declare`** statement. We assume that the user feeds interactive statements to the system one by one. (In the examples given throughout the book, the **`declare`** statement is often left out. It should be added if the example declares new variables.)

The interactive interface allows much more than just feeding statements. It has all the functionality needed for software development. Appendix A gives a summary of some of this functionality. For now, we assume that the user just knows how to feed statements.

The interactive interface has a single, global environment. The **`declare`** statement adds new mappings to this environment. It follows that **`declare`** can *only* be used interactively, not in standalone programs. Feeding the following declaration:

```
declare X Y
```

creates two new variables in the store, x_1 and x_2, and adds mappings from `X` and `Y` to them. Because the mappings are in the global environment we say that `X` and `Y` are global variables or interactive variables. Feeding the same declaration a second time will cause `X` and `Y` to map to two other new variables, x_3 and x_4. Figure 2.19 shows what happens. The original variables, x_1 and x_2, are still in the store, but they are no longer referred to by `X` and `Y`. In the figure, `Browse` maps to a procedure value that implements the browser. The **`declare`** statement adds new variables and mappings, but leaves existing variables in the store unchanged.

Adding a new mapping to an identifier that already maps to a variable may

Figure 2.20: The Browser.

cause the variable to become inaccessible if there are no other references to it. If the variable is part of a calculation, then it is still accessible from within the calculation. For example:

```
declare X Y
X=25
declare A
A=person(age:X)
declare X Y
```

Just after the binding `X=25`, `X` maps to 25, but after the second **`declare`** `X Y` it maps to a new unbound variable. The 25 is still accessible through the global variable `A`, which is bound to the record `person(age:25)`. The record contains 25 because `X` mapped to 25 when the binding `A=person(age:X)` was executed. The second **`declare`** `X Y` changes the mapping of `X`, but not the record `person(age:25)` since the record already exists in the store. This behavior of **`declare`** is designed to support a modular programming style. Executing a program fragment will not cause the results of any previously executed fragment to change.

There is a second form of **`declare`**:

`declare` `X Y` **`in`** ⟨stmt⟩

which declares two global variables, as before, and then executes ⟨stmt⟩. The difference with the first form is that ⟨stmt⟩ declares no global variables (unless it contains a **`declare`**).

The Browser

The interactive interface has a tool, called the *Browser*, which allows looking into the store. This tool is available to the programmer as a procedure called `Browse`.

the problem is a blocked operation. Carefully check all operations to make sure that their arguments are bound. Ideally, the system's debugger should detect when a program has blocked operations that cannot continue.

2.7 Exceptions

First let us find the rule, then we will try to explain the exceptions.
– *The Name of the Rose*, Umberto Eco (1932–)

How do we handle exceptional situations within a program? For example, dividing by zero, opening a nonexistent file, or selecting a nonexistent field of a record? These operations do not occur in a correct program, so they should not encumber normal programming style. On the other hand, they do occur sometimes. It should be possible for programs to manage them in a simple way. The declarative model cannot do this without adding cumbersome checks throughout the program. A more elegant way is to extend the model with an exception-handling mechanism. This section does exactly that. We give the syntax and semantics of the extended model and explain what exceptions look like in the full language.

2.7.1 Motivation and basic concepts

In the semantics of section 2.4, we speak of "raising an error" when a statement cannot continue correctly. For example, a conditional raises an error when its argument is a non-boolean value. Up to now, we have been deliberately vague about exactly what happens next. Let us now be more precise. We define an error as a difference between the actual behavior of a program and its desired behavior. There are many sources of errors, both internal and external to the program. An internal error could result from invoking an operation with an argument of illegal type or illegal value. An external error could result from opening a nonexistent file.

We would like to be able to detect errors and handle them from within a running program. The program should not stop when they occur. Rather, it should in a controlled way transfer execution to another part, called the exception handler, and pass the exception handler a value that describes the error.

What should the exception-handling mechanism look like? We can make two observations. First, it should be able to confine the error, i.e., quarantine it so that it does not contaminate the whole program. We call this the error confinement principle. Assume that the program is made up of interacting "components" organized in hierarchical fashion. Each component is built of smaller components. We put "component" in quotes because the language does not need to have a component concept. It just needs to be compositional, i.e., programs are built in layered fashion. Then the error confinement principle states that an error in a component should be catchable at the component boundary. Outside the component, the error is either invisible or reported in a nice way.

exception can be raised (and caught) before it is known which exception it is! This is quite reasonable in a language with dataflow variables: we may at some point know that there exists a problem but not know yet which problem.

An example

Let us give a simple example of exception handling. Consider the following function, which evaluates simple arithmetic expressions and returns the result:

```
fun {Eval E}
   if {IsNumber E} then E
   else
      case E
      of   plus(X Y) then {Eval X}+{Eval Y}
      []   times(X Y) then {Eval X}*{Eval Y}
      else raise illFormedExpr(E) end
      end
   end
end
```

For this example, we say an expression is ill-formed if it is not recognized by `Eval`, i.e., if it contains other values than numbers, `plus`, and `times`. Trying to evaluate an ill-formed expression `E` will raise an exception. The exception is a tuple, `illFormedExpr(E)`, that contains the ill-formed expression. Here is an example of using `Eval`:

```
try
   {Browse {Eval plus(plus(5 5) 10)}}
   {Browse {Eval times(6 11)}}
   {Browse {Eval minus(7 10)}}
catch illFormedExpr(E) then
   {Browse ´*** Illegal expression ´#E#´ ***´}
end
```

If any call to `Eval` raises an exception, then control transfers to the **`catch`** clause, which displays an error message.

2.7.2 The declarative model with exceptions

We extend the declarative computation model with exceptions. Table 2.9 gives the syntax of the extended kernel language. Programs can use two new statements, **`try`** and **`raise`**. In addition, there is a third statement, **`catch`** ⟨x⟩ **`then`** ⟨s⟩ **`end`**, that is needed internally for the semantics and is not allowed in programs. The **`catch`** statement is a "marker" on the semantic stack that defines the boundary of the exception-catching context. We now give the semantics of these statements.

The `try` *statement*

The semantic statement is:

⟨s⟩ ::=	
skip	Empty statement
⟨s⟩$_1$ ⟨s⟩$_2$	Statement sequence
local ⟨x⟩ **in** ⟨s⟩ **end**	Variable creation
⟨x⟩$_1$=⟨x⟩$_2$	Variable-variable binding
⟨x⟩=⟨v⟩	Value creation
if ⟨x⟩ **then** ⟨s⟩$_1$ **else** ⟨s⟩$_2$ **end**	Conditional
case ⟨x⟩ **of** ⟨pattern⟩ **then** ⟨s⟩$_1$ **else** ⟨s⟩$_2$ **end**	Pattern matching
{⟨x⟩ ⟨y⟩$_1$ ⋯ ⟨y⟩$_n$}	Procedure application
try ⟨s⟩$_1$ **catch** ⟨x⟩ **then** ⟨s⟩$_2$ **end**	**Exception context**
raise ⟨x⟩ **end**	**Raise exception**

Table 2.9: The declarative kernel language with exceptions.

(**try** ⟨s⟩$_1$ **catch** ⟨x⟩ **then** ⟨s⟩$_2$ **end**, E)

Execution consists of the following actions:

- Push the semantic statement (**catch** ⟨x⟩ **then** ⟨s⟩$_2$ **end**, E) on the stack.
- Push ($\langle s \rangle_1, E$) on the stack.

The `raise` *statement*

The semantic statement is:

(**raise** ⟨x⟩ **end**, E)

Execution consists of the following actions:

- Pop elements off the stack looking for a **catch** statement.
  - If a **catch** statement is found, pop it from the stack.
  - If the stack is emptied and no **catch** is found, then stop execution with the error message "`Uncaught exception`".
- Let (**catch** ⟨y⟩ **then** ⟨s⟩ **end**, E_c) be the **catch** statement that is found.
- Push ($\langle s \rangle, E_c + \{\langle y \rangle \rightarrow E(\langle x \rangle)\}$) on the stack.

Let us see how an uncaught exception is handled by the Mozart system. For interactive execution, an error message is printed in the Oz emulator window. For standalone applications, the application terminates and an error message is sent on the standard error output of the process. It is possible to change this behavior to something else that is more desirable for particular applications, by using the System module `Property`.

⟨statement⟩	::=	**try** ⟨inStatement⟩
		[**catch** ⟨pattern⟩ **then** ⟨inStatement⟩
		{ ´[]´ ⟨pattern⟩ **then** ⟨inStatement⟩ }]
		[**finally** ⟨inStatement⟩] **end**
	|	**raise** ⟨inExpression⟩ **end**
	|	...
⟨inStatement⟩	::=	[{ ⟨declarationPart⟩ }+ **in**] ⟨statement⟩
⟨inExpression⟩	::=	[{ ⟨declarationPart⟩ }+ **in**] [⟨statement⟩] ⟨expression⟩

Table 2.10: Exception syntax.

The `catch` *statement*

The semantic statement is:

(**catch** ⟨x⟩ **then** ⟨s⟩ **end**, E)

Execution is complete after this pair is popped from the semantic stack. That is, the **catch** statement does nothing, just like **skip**.

2.7.3 Full syntax

Table 2.10 gives the syntax of the **try** statement in the full language. It has an optional **finally** clause. The **catch** clause has an optional series of patterns. Let us see how these extensions are defined.

The `finally` *clause*

A **try** statement can specify a **finally** clause which is always executed, whether or not the statement raises an exception. The new syntax

try $\langle s \rangle_1$ **finally** $\langle s \rangle_2$ **end**

is translated to the kernel language as:

```
try ⟨s⟩1
catch X then
   ⟨s⟩2
   raise X end
end
⟨s⟩2
```

(where an identifier X is chosen that is not free in $\langle s \rangle_2$). It is possible to define a translation in which $\langle s \rangle_2$ only occurs once; we leave this to the exercises.

The **finally** clause is useful when dealing with entities that are external to the computation model. With **finally**, we can guarantee that some "cleanup"

action gets performed on the entity, whether or not an exception occurs. A typical example is reading a file. Assume `F` is an open file,[12] the procedure `ProcessFile` manipulates the file in some way, and the procedure `CloseFile` closes the file. Then the following program ensures that `F` is always closed after `ProcessFile` completes, whether or not an exception was raised:

```
try
   {ProcessFile F}
finally {CloseFile F} end
```

Note that this **try** statement does not catch the exception; it just executes `CloseFile` whenever `ProcessFile` completes. We can combine both catching the exception and executing a final statement:

```
try
   {ProcessFile F}
catch X then
   {Browse ´*** Exception ´#X#´ when processing file ***´}
finally {CloseFile F} end
```

This behaves like two nested **try** statements: the innermost with just a **catch** clause and the outermost with just a **finally** clause.

Pattern matching

A **try** statement can use pattern matching to catch only exceptions that match a given pattern. Other exceptions are passed to the next enclosing **try** statement. The new syntax:

```
try ⟨s⟩
catch ⟨p⟩1 then ⟨s⟩1
   [] ⟨p⟩2 then ⟨s⟩2
   ...
   [] ⟨p⟩n then ⟨s⟩n
end
```

is translated to the kernel language as:

```
try ⟨s⟩
catch X then
   case X
   of ⟨p⟩1 then ⟨s⟩1
   [] ⟨p⟩2 then ⟨s⟩2
   ...
   [] ⟨p⟩n then ⟨s⟩n
   else raise X end
   end
end
```

If the exception does not match any of the patterns, then it is simply raised again.

12. We will see later how file input/output is handled.

The λ calculus

Pure functional languages are based on a formalism called the λ calculus. There are many variants of the λ calculus. All of these variants have in common two basic operations, namely defining and evaluating functions. For example, the function value `fun {$ X} X*X end` is identical to the λ expression $\lambda x.\ x*x$. This expression consists of two parts: the x before the dot, which is the function's argument, and the expression $x * x$, which is the function's result. The `Append` function, which appends two lists together, can be defined as a function value:

```
Append=fun {$ Xs Ys}
          if {IsNil Xs} then Ys
          else {Cons {Car Xs} {Append {Cdr Xs} Ys}}
          end
       end
```

This is equivalent to the following λ expression:

$$append = \lambda xs, ys\ .\ \text{if } isNil(xs) \text{ then } ys \\ \text{else } cons(car(xs), append(cdr(xs), ys))$$

This definition of `Append` uses the following helper functions:

```
fun {IsNil X} X==nil end
fun {IsCons X} case X of _|_ then true else false end end
fun {Car H|T} H end
fun {Cdr H|T} T end
fun {Cons H T} H|T end
```

Restricting the declarative model

The declarative model is more general than the λ calculus in two ways. First, it defines functions on partial values, i.e., with unbound variables. Second, it uses a procedural syntax. We can define a pure functional language by putting two syntactic restrictions on the declarative model so that it always calculates functions on complete values:

- Always bind a variable to a value immediately when it is declared. That is, the `local` statement always has one of the following two forms:

```
local ⟨x⟩=⟨v⟩ in ⟨s⟩ end
local ⟨x⟩={⟨y⟩ ⟨y⟩1 ... ⟨y⟩n} in ⟨s⟩ end
```

- Use only the function syntax, not the procedure syntax. For function calls inside data structures, do the nested call before creating the data structure (instead of after, as in section 2.6.2). This avoids putting unbound variables in data structures.

With these restrictions, the model no longer needs unbound variables. The declarative model with these restrictions is called the (strict) functional model. This model is close to well-known functional programming languages such as Scheme and Standard ML. The full range of higher-order programming techniques is pos-

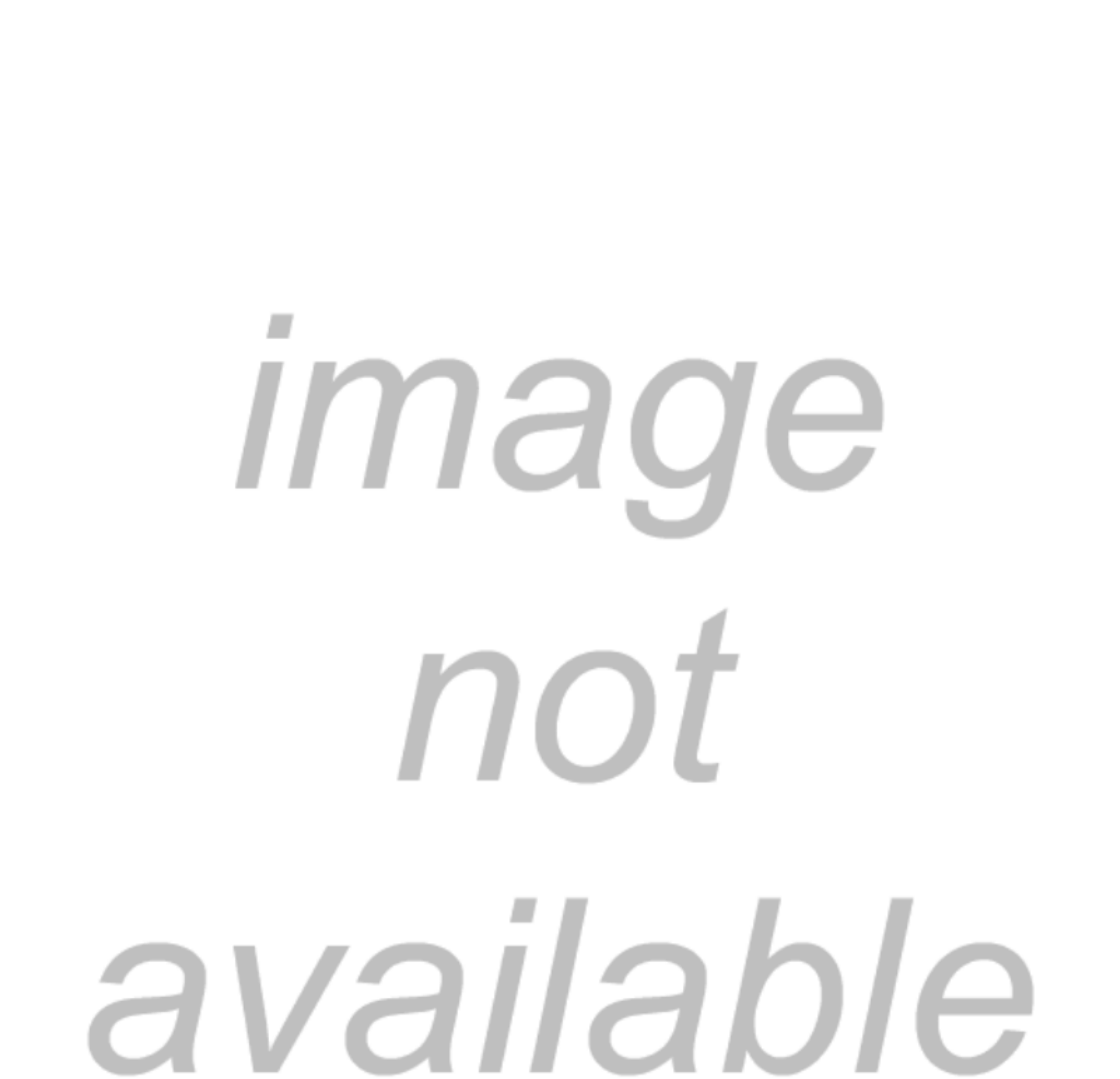


image
not
available

⟨statement⟩	::=	⟨expression⟩ "=" ⟨expression⟩ | ...
⟨expression⟩	::=	⟨expression⟩ "==" ⟨expression⟩
	|	⟨expression⟩ "\=" ⟨expression⟩ | ...
⟨binaryOp⟩	::=	"=" | "==" | "\=" | ...

Table 2.11: Equality (unification) and equality test (entailment check).

now examine the general cases.

Binding a variable to a value is a special case of an operation called unification. The unification ⟨Term1⟩=⟨Term2⟩ makes the partial values ⟨Term1⟩ and ⟨Term2⟩ equal, if possible, by adding zero or more bindings to the store. For example, `f(X Y)=f(1 2)` does two bindings: `X=1` and `Y=2`. If the two terms cannot be made equal, then an exception is raised. Unification exists because of partial values; if there would be only complete values, then it would have no meaning.

Testing whether a variable is equal to a value is a special case of the entailment check and disentailment check operations. The entailment check ⟨Term1⟩==⟨Term2⟩ (and its opposite, the disentailment check ⟨Term1⟩\=⟨Term2⟩) is a two-argument boolean function that blocks until it is known whether ⟨Term1⟩ and ⟨Term2⟩ are equal or not equal.[14] Entailment and disentailment checks never do any binding.

2.8.2.1 *Unification (the = operation)*

A good way to conceptualize unification is as an operation that adds information to the single-assignment store. The store is a set of dataflow variables, where each variable is either unbound or bound to some other store entity. The store's information is just the set of all its bindings. Doing a new binding, e.g., `X=Y`, will add the information that `X` and `Y` are equal. If `X` and `Y` are already bound when doing `X=Y`, then some other bindings may be added to the store. For example, if the store already has `X=foo(A)` and `Y=foo(25)`, then doing `X=Y` will bind `A` to 25. Unification is a kind of "compiler" that is given new information and "compiles it into the store," taking into account the bindings that are already there. To understand how this works, let us look at some possibilities.

- The simplest cases are bindings to values, e.g., `X=person(name:X1 age:X2)`, and variable-variable bindings, e.g., `X=Y`. If `X` and `Y` are unbound, then these operations each add one binding to the store.
- Unification is symmetric. For example, `person(name:X1 age:X2)=X` means the

14. The word "entailment" comes from logic. It is a form of logical implication. This is because the equality ⟨Term1⟩==⟨Term2⟩ is true if the store, considered as a conjunction of equalities, "logically implies" ⟨Term1⟩==⟨Term2⟩.

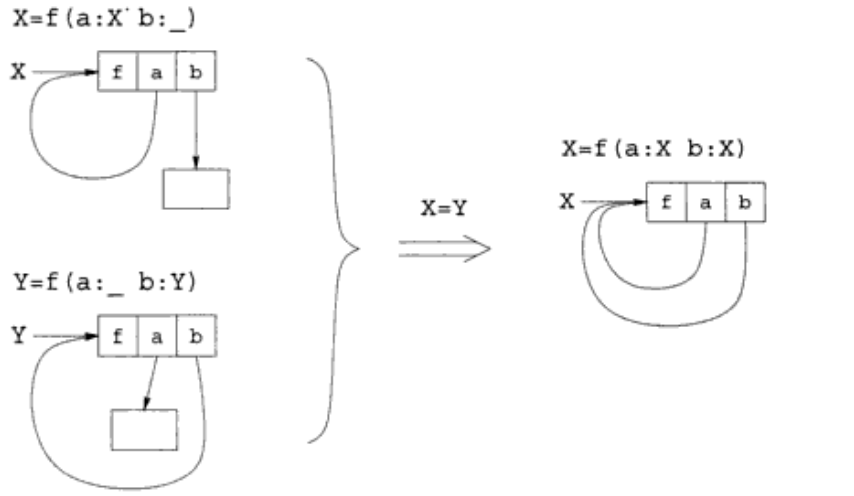


Figure 2.22: Unification of cyclic structures.

same as `X=person(name:X1 age:X2)`.

- Any two partial values can be unified. For example, unifying the two records:

```
person(name:X1 age:X2)
person(name:"George" age:25)
```

This binds `X1` to `"George"` and `X2` to 25.

- If the partial values are already equal, then unification does nothing. For example, unifying `X` and `Y` where the store contains the two records:

```
X=person(name:"George" age:25)
Y=person(name:"George" age:25)
```

This does nothing.

- If the partial values are incompatible, then they cannot be unified. For example, unifying the two records:

```
person(name:X1 age:26)
person(name:"George" age:25)
```

The records have different values for their `age` fields, namely 25 and 26, so they cannot be unified. This unification will raise a `failure` exception, which can be caught by a **`try`** statement. The unification might or might not bind `X1` to `"George"`; it depends on exactly when it finds out that there is an incompatibility. Another way to get a unification failure is by executing the statement **`fail`**.

- Unification is symmetric in the arguments. For example, unifying the two records:

```
person(name:"George" age:X2)
person(name:X1 age:25)
```

This binds `X1` to `"George"` and `X2` to 25, just like before.

- Unification can create cyclic structures, i.e., structures that refer to themselves. For example, the unification `X=person(grandfather:X)`. This creates a record whose `grandfather` field refers to itself. This situation happens in some crazy time-travel stories.
- Unification can bind cyclic structures. For example, let's create two cyclic structures, in `X` and `Y`, by doing `X=f(a:X b:_)` and `Y=f(a:_ b:Y)`. Now, doing the unification `X=Y` creates a structure with two cycles, which we can write as `X=f(a:X b:X)`. This example is illustrated in figure 2.22.

2.8.2.2 The unification algorithm

Let us give a precise definition of unification. We will define the operation unify(x, y) that unifies two partial values x and y in the store σ. Unification is a basic operation of logic programming. When used in the context of unification, store variables are called logic variables. Logic programming, which is also called relational programming, is discussed in chapter 9.

The store The store consists of a set of k variables, $x_1, \ldots, x_k$, that are partitioned as follows:

- Sets of unbound variables that are equal (also called equivalence sets of variables). The variables in each set are equal to each other but not to any other variables.
- Variables bound to a number, record, or procedure (also called determined variables).

An example is the store $\{x_1 = \texttt{foo(a:}x_2\texttt{)}, x_2 = 25, x_3 = x_4 = x_5, x_6, x_7 = x_8\}$ that has eight variables. It has three equivalence sets, namely $\{x_3, x_4, x_5\}$, $\{x_6\}$, and $\{x_7, x_8\}$. It has two determined variables, namely x_1 and x_2.

The primitive bind operation We define unification in terms of a primitive bind operation on the store σ. The operation binds all variables in an equivalence set:

- bind$(ES, \langle \mathsf{v} \rangle)$ binds all variables in the equivalence set ES to the number or record $\langle \mathsf{v} \rangle$. For example, the operation bind$(\{x_7, x_8\}, \texttt{foo(a:}x_2\texttt{)})$ modifies the example store so that x_7 and x_8 are no longer in an equivalence set but both become bound to `foo(a:`x_2`)`.
- bind(ES_1, ES_2) merges the equivalence set ES_1 with the equivalence set ES_2. For example, the operation bind$(\{x_3, x_4, x_5\}, \{x_6\})$ modifies the example store so that x_3, x_4, x_5, and x_6 are in a single equivalence set, namely $\{x_3, x_4, x_5, x_6\}$.

The algorithm We now define the operation unify(x, y) as follows:

1. If x is in the equivalence set ES_x and y is in the equivalence set ES_y, then do bind(ES_x, ES_y). If x and y are in the same equivalence set, this is the same as

doing nothing.

2. If x is in the equivalence set ES_x and y is determined, then do bind(ES_x, y).

3. If y is in the equivalence set ES_y and x is determined, then do bind(ES_y, x).

4. If x is bound to $l(l_1 : x_1, \ldots, l_n : x_n)$ and y is bound to $l'(l'_1 : y_1, \ldots, l'_m : y_m)$ with $l \neq l'$ or $\{l_1, \ldots, l_n\} \neq \{l'_1, \ldots, l'_m\}$, then raise a failure exception.

5. If x is bound to $l(l_1 : x_1, \ldots, l_n : x_n)$ and y is bound to $l(l_1 : y_1, \ldots, l_n : y_n)$, then for i from 1 to n do unify(x_i, y_i).

Handling cycles The above algorithm does not handle unification of partial values with cycles. For example, assume the store contains x = `f(a:x)` and y = `f(a:y)`. Calling unify(x, y) results in the recursive call unify(x, y), which is identical to the original call. The algorithm loops forever! Yet it is clear that x and y have exactly the same structure: what the unification *should* do is add exactly zero bindings to the store and then terminate. How can we fix this problem?

A simple fix is to make sure that unify(x, y) is called at most once for each possible pair of two variables (x, y). Since any attempt to call it again will not do anything new, it can return immediately. With k variables in the store, this means at most k^2 unify calls, so the algorithm is guaranteed to terminate. In practice, the number of unify calls is much less than this. We can implement the fix with a table that stores all called pairs. This gives the new algorithm unify$'(x, y)$:

- Let M be a new, empty table.
- Call unify$''(x, y)$.

This needs the definition of unify$''(x, y)$:

- If $(x, y) \in M$, then we are done.
- Otherwise, insert (x, y) in M and then do the original algorithm for unify(x, y), in which the recursive calls to unify are replaced by calls to unify$''$.

This algorithm can be written in the declarative model by passing M as two extra arguments to unify$''$. A table that remembers previous calls so that they can be avoided in the future is called a memoization table.

2.8.2.3 Displaying cyclic structures

We have seen that unification can create cyclic structures. To display these in the browser, it has to be configured right. In the browser's `Options` menu, pick the `Representation` entry and choose the `Graph` mode. There are three display modes, namely `Tree` (the default), `Graph`, and `Minimal Graph`. `Tree` does not take sharing or cycles into account. `Graph` correctly handles sharing and cycles by displaying a graph. `Minimal Graph` shows the smallest graph that is consistent with the data. We give some examples. Consider the following two unifications:

```
declare L1 L2 L3 Head Tail in
L1=Head|Tail
Head=1
Tail=2|nil

L2=[1 2]
{Browse L1==L2}

L3='|'(1:1 2:'|'(2 nil))
{Browse L1==L3}
```

All three lists, `L1`, `L2`, and `L3`, are identical. Here is an example where the entailment check cannot decide:

```
declare L1 L2 X in
L1=[1]
L2=[X]
{Browse L1==L2}
```

Feeding this example will not display anything, since the entailment check cannot decide whether `L1` and `L2` are equal or not. In fact, both are possible: if `X` is bound to `1`, then they are equal, and if `X` is bound to `2`, then they are not. Try feeding `X=1` or `X=2` to see what happens. What about the following example?:

```
declare L1 L2 X in
L1=[X]
L2=[X]
{Browse L1==L2}
```

Both lists contain the same unbound variable `X`. What will happen? Think about it before reading the answer in the footnote.[16] Here is a final example:

```
declare L1 L2 X in
L1=[1 a]
L2=[X b]
{Browse L1==L2}
```

This will display **`false`**. While the comparison `1==X` blocks, further inspection of the two graphs shows that there is a definite difference, so the full check returns **`false`**.

2.8.3 Dynamic and static typing

The only way of discovering the limits of the possible is to venture a little way past them into the impossible.
– Clarke's second law, Arthur C. Clarke (1917–)

It is important for a language to be strongly typed, i.e., to have a type system that is enforced by the language. (This is in contrast to a weakly typed language, in which the internal representation of a type can be manipulated by a program. We

16. The browser will display **`true`**, since `L1` and `L2` are equal no matter what `X` might be bound to.

efficient representation. For example, if a variable is of boolean type, the compile can implement it with a single bit. In a dynamically typed language, the compiler cannot always deduce the type of a variable. When it cannot, then it usually has to allocate a full memory word, so that any possible value (or a pointer to a value) can be accommodated.

- Static typing can improve the security of a program. Secure data abstractions can be constructed based solely on the protection offered by the type system.

Unfortunately, the choice between dynamic and static typing is most often based on emotional ("gut") reactions, not on rational argument. Adherents of dynamic typing relish the expressive freedom and rapid turnaround it gives them and criticize the reduced expressiveness of static typing. On the other hand, adherents of static typing emphasize the aid it gives them in writing correct and efficient programs and point out that it finds many program errors at compile time. Little hard data exist to quantify these differences. In our experience, the differences are not great. Programming with static typing is like word processing with a spelling checker: a good writer can get along without it, but it can improve the quality of a text.

Each approach has a role in practical application development. Static typing is recommended when the programming techniques are well understood and when efficiency and correctness are paramount. Dynamic typing is recommended for rapid development and when programs must be as flexible as possible, such as application prototypes, operating systems, and some artificial intelligence applications.

The choice between static and dynamic typing does not have to be all-or-nothing. In each approach, a bit of the other can be added, gaining some of its advantages. For example, different kinds of polymorphism (where a variable might have values of several different types) add flexibility to statically typed functional and object-oriented languages. It is an active research area to design static type systems that capture as much as possible of the flexibility of dynamic type systems, while encouraging good programming style and still permitting compile time verification.

The computation models given in the book are all subsets of the Oz language, which is dynamically typed. One research goal of the Oz project is to explore what programming techniques are possible in a computation model that integrates several programming paradigms. The only way to achieve this goal is with dynamic typing.

When the programming techniques are known, then a possible next step is to design a static type system. While research into increasing the functionality and expressiveness of Oz is still ongoing in the Mozart Consortium, the Alice project at Saarland University in Saarbrücken, Germany, has chosen to add a static type system. Alice is a statically typed language that has much of the expressiveness of Oz. At the time of writing, Alice is interoperable with Oz (programs can be written partly in Alice and partly in Oz) since it is based on the Mozart implementation.

2.9 Exercises

1. *Free and bound identifiers.* Consider the following statement:

```
proc {P X}
   if X>0 then {P X-1} end
end
```

Is the second occurrence of the identifier `P` free or bound? Justify your answer. *Hint*: this is easy to answer if you first translate to kernel syntax.

2. *Contextual environment.* Section 2.4 explains how a procedure call is executed. Consider the following procedure `MulByN`:

```
declare MulByN N in
N=3
proc {MulByN X ?Y}
   Y=N*X
end
```

together with the call `{MulByN A B}`. Assume that the environment at the call contains $\{\texttt{A} \rightarrow 10, \texttt{B} \rightarrow x_1\}$. When the procedure body is executed, the mapping `N` $\rightarrow 3$ is added to the environment. Why is this a necessary step? In particular, would not `N` $\rightarrow 3$ already exist somewhere in the environment at the call? Would not this be enough to ensure that the identifier `N` already maps to `3`? Give an example where `N` does not exist in the environment at the call. Then give a second example where `N` does exist there, but is bound to a different value than `3`.

3. *Functions and procedures.* If a function body has an **`if`** statement with a missing **`else`** case, then an exception is raised if the **`if`** condition is false. Explain why this behavior is correct. This situation does not occur for procedures. Explain why not.

4. *The* **`if`** *and* **`case`** *statements.* This exercise explores the relationship between the **`if`** statement and the **`case`** statement.

   (a) Define the **`if`** statement in terms of the **`case`** statement. This shows that the conditional does not add any expressiveness over pattern matching. It could have been added as a linguistic abstraction.

   (b) Define the **`case`** statement in terms of the **`if`** statement, using the operations `Label`, `Arity`, and `'.'` (feature selection).

This shows that the **`if`** statement is essentially a more primitive version of the **`case`** statement.

5. *The* **`case`** *statement.* This exercise tests your understanding of the full **`case`** statement. Given the following procedure:

```
proc {Test X}
   case X
   of a|Z then {Browse 'case'(1)}
   [] f(a) then {Browse 'case'(2)}
   [] Y|Z andthen Y==Z then {Browse 'case'(3)}
   [] Y|Z then {Browse 'case'(4)}
   [] f(Y) then {Browse 'case'(5)}
   else {Browse 'case'(6)} end
end
```

Without executing any code, predict what will happen when you feed {Test [b c a]}, {Test f(b(3))}, {Test f(a)}, {Test f(a(3))}, {Test f(d)}, {Test [a b c]}, {Test [c a b]}, {Test a|a}, and {Test '|'(a b c)}. Use the kernel translation and the semantics if necessary to make the predictions. After making the predictions, check your understanding by running the examples in Mozart.

6. *The* **case** *statement again.* Given the following procedure:

```
proc {Test X}
   case X of f(a Y c) then {Browse 'case'(1)}
   else {Browse 'case'(2)} end
end
```

Without executing any code, predict what will happen when you feed:

```
declare X Y {Test f(X b Y)}
```

Same for:

```
declare X Y {Test f(a Y d)}
```

Same for:

```
declare X Y {Test f(X Y d)}
```

Use the kernel translation and the semantics if necessary to make the predictions. After making the predictions, check your understanding by running the examples in Mozart. Now run the following example:

```
declare X Y
if f(X Y d)==f(a Y c) then {Browse 'case'(1)}
else {Browse 'case'(2)} end
```

Does this give the same result or a different result than the previous example? Explain the result.

7. *Lexically scoped closures.* Given the following code:

```
declare Max3 Max5
proc {SpecialMax Value ?SMax}
   fun {SMax X}
      if X>Value then X else Value end
   end
end
{SpecialMax 3 Max3}
{SpecialMax 5 Max5}
```

Without executing any code, predict what will happen when you feed:

```
{Browse [{Max3 4} {Max5 4}]}
```

Check your understanding by running this example in Mozart.

8. *Control abstraction.* This exercise explores the relationship between linguistic abstractions and higher-order programming.

(a) Define the function `AndThen` as follows:

```
fun {AndThen BP1 BP2}
   if {BP1} then {BP2} else false end
end
```

Does the call

{`AndThen` **fun** {`$`} ⟨expression⟩$_1$ **end fun** {`$`} ⟨expression⟩$_2$ **end**}

give the same result as ⟨expression⟩$_1$ **andthen** ⟨expression⟩$_2$? Does it avoid the evaluation of ⟨expression⟩$_2$ in the same situations?

(b) Write a function `OrElse` that is to **orelse** as `AndThen` is to **andthen**. Explain its behavior.

9. *Tail recursion.* This exercise examines the importance of tail recursion, in the light of the semantics given in the chapter. Consider the following two functions:

```
fun {Sum1 N}
   if N==0 then 0 else N+{Sum1 N-1} end
end
fun {Sum2 N S}
   if N==0 then S else {Sum2 N-1 N+S} end
end
```

Now do the following:

(a) Expand the two definitions into kernel syntax. It should be clear that `Sum2` is tail recursive and `Sum1` is not.

(b) Execute the two calls `{Sum1 10}` and `{Sum2 10 0}` by hand, using the semantics of this chapter to follow what happens to the stack and the store. How large does the stack become in either case?

(c) What would happen in the Mozart system if you would call `{Sum1 100000000}` or `{Sum2 100000000 0}`? Which one is likely to work? Which one is not? Try both on Mozart to verify your reasoning.

10. *Expansion into kernel syntax.* Consider the following function `SMerge` that merges two sorted lists:

```
fun {SMerge Xs Ys}
   case Xs#Ys
   of nil#Ys then Ys
   [] Xs#nil then Xs
   [] (X|Xr)#(Y|Yr) then
      if X=<Y then X|{SMerge Xr Ys}
      else Y|{SMerge Xs Yr} end
   end
end
```

Expand `SMerge` into the kernel syntax. Note that `X#Y` is a tuple of two arguments

3 Declarative Programming Techniques

S'il vous plaît... dessine-moi un arbre!
If you please—draw me a tree!
– Freely adapted from *Le Petit Prince*, Antoine de Saint-Exupéry (1900–1944)

The nice thing about declarative programming is that you can write a specification and run it as a program. The nasty thing about declarative programming is that some clear specifications make incredibly bad programs. The hope of declarative programming is that you can move from a specification to a reasonable program without leaving the language.
– *The Craft of Prolog*, Richard O'Keefe (1990)

Consider any computational operation, i.e., a program fragment with inputs and outputs. We say the operation is declarative if, whenever called with the same arguments, it returns the same results independent of any other computation state. Figure 3.1 illustrates the concept. A declarative operation is independent (does not depend on any execution state outside of itself), stateless (has no internal execution state that is remembered between calls), and deterministic (always gives the same results when given the same arguments). We will show that all programs written using the computation model of the last chapter are declarative.

Why declarative programming is important

Declarative programming is important because of two properties:

- *Declarative programs are compositional.* A declarative program consists of components that can each be written, tested, and proved correct independently of other components and of its own past history (previous calls).
- *Reasoning about declarative programs is simple.* Programs written in the declarative model are easier to reason about than programs written in more expressive models. Since declarative programs compute only values, simple algebraic and logical reasoning techniques can be used.

These two properties are important both for programming in the large and in the small. It would be nice if all programs could easily be written in the declarative model. Unfortunately, this is not the case. The declarative model is a good fit for certain kinds of programs and a bad fit for others. This chapter and the next

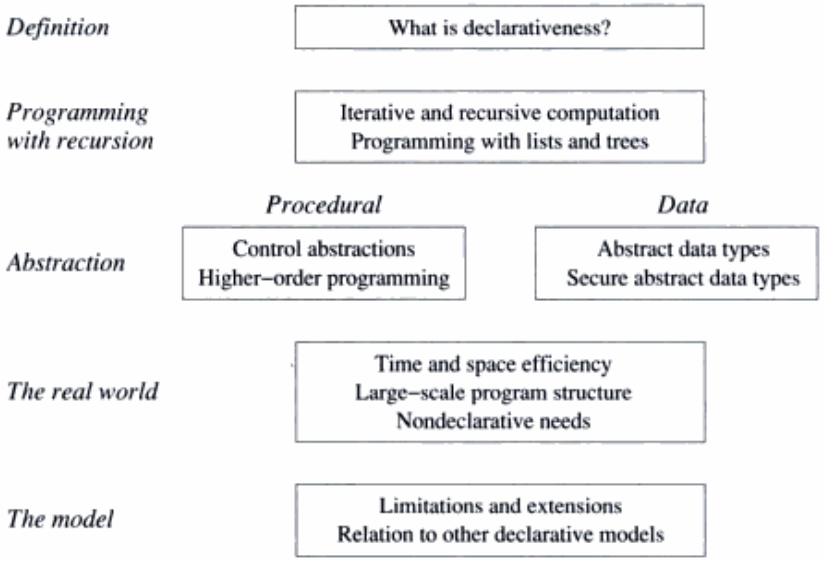


Figure 3.2: Structure of the chapter.

Writing declarative programs

The simplest way to write a declarative program is to use the declarative model of chapter 2. The basic operations on data types are declarative, e.g., the arithmetic, list, and record operations. It is possible to combine declarative operations to make new declarative operations, if certain rules are followed. Combining declarative operations according to the operations of the declarative model will result in a declarative operation. This is explained in section 3.1.3.

The standard rule in algebra that "equals can be replaced by equals" is another example of a declarative combination. In programming languages, this property is called referential transparency. It greatly simplifies reasoning about programs. For example, if we know that $f(a) = a^2$, then we can replace $f(a)$ by a^2 in any other place where it occurs. The equation $b = 7f(a)^2$ then becomes $b = 7a^4$. This is possible because $f(a)$ is declarative: it depends only on its arguments and not on any other computation state.

The basic technique for writing declarative programs is to consider the program as a set of recursive function definitions, using higher-order programming to simplify the program structure. A recursive function is one whose definition body refers to the function itself, either directly or indirectly. Direct recursion means that the function itself is used in the body. Indirect recursion means that the function refers to another function that directly or indirectly refers to the original function. Higher-order programming means that functions can have other functions as arguments and results. This ability underlies all the techniques for building abstractions that we

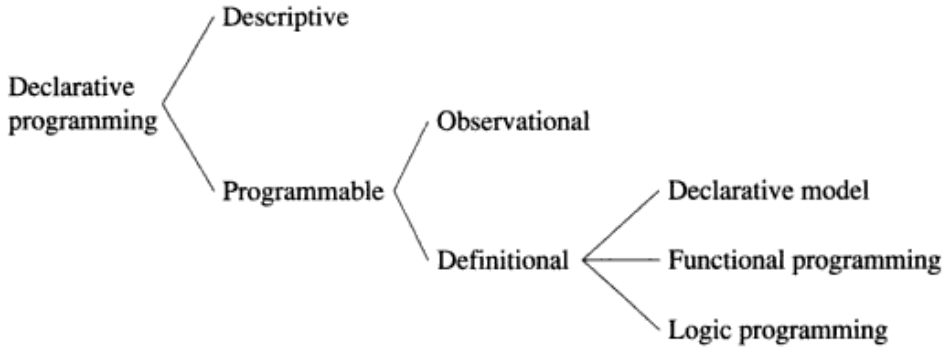


Figure 3.3: A classification of declarative programming.

will show in the book. Higher-orderness can compensate somewhat for the lack of expressiveness of the declarative model, i.e., it makes it easy to code limited forms of concurrency and state in the declarative model.

Structure of the chapter

This chapter explains how to write practical declarative programs. The chapter is roughly organized into the six parts shown in figure 3.2. The first part defines "declarativeness." The second part gives an overview of programming techniques. The third and fourth parts explain procedural and data abstraction. The fifth part shows how declarative programming interacts with the rest of the computing environment. The sixth part steps back to reflect on the usefulness of the declarative model and situate it with respect to other models.

3.1 What is declarativeness?

The declarative model of chapter 2 is an especially powerful way of writing declarative programs, since all programs written in it will be declarative by this fact alone. But it is still only one way out of many for doing declarative programming. Before explaining how to program in the declarative model, let us situate it with respect to the other ways of being declarative. Let us also explain why programs written in it are always declarative.

3.1.1 A classification of declarative programming

We have defined declarativeness in one particular way, so that reasoning about programs is simplified. But this is not the only way to make precise what declarative programming is. Intuitively, it is programming by defining the what (the results

⟨s⟩ ::=		
	skip	Empty statement
|	⟨s⟩$_1$ ⟨s⟩$_2$	Statement sequence
|	**local** ⟨x⟩ **in** ⟨s⟩ **end**	Variable creation
|	⟨x⟩$_1$=⟨x⟩$_2$	Variable-variable binding
|	⟨x⟩=⟨v⟩	Value creation

Table 3.1: The descriptive declarative kernel language.

we want to achieve) without explaining the how (the algorithms, etc., needed to achieve the results). This vague intuition covers many different ideas. Figure 3.3 gives a classification. The first level of classification is based on the expressiveness. There are two possibilities:

- A descriptive declarativeness. This is the least expressive. The declarative "program" just defines a data structure. Table 3.1 defines a language at this level. This language can only define records! It contains just the first five statements of the kernel language in table 2.1. Section 3.8.2 shows how to use this language to define graphical user interfaces. Other examples are a formatting language like HTML (Hypertext Markup Language), which gives the structure of a document without telling how to do the formatting, or an information exchange language like XML (Extensible Markup Language), which is used to exchange information in an open format that is easily readable by all. The descriptive level is too weak to write general programs. So why is it interesting? Because it consists of data structures that are easy to calculate with. The records of table 3.1, HTML and XML documents, and the declarative user interfaces of section 3.8.2 can all be created and transformed easily by a program.
- A programmable declarativeness. This is as expressive as a Turing machine.[1] For example, table 2.1 defines a language at this level. See the introduction to chapter 6 for more on the relationship between the descriptive and programmable levels.

There are two fundamentally different ways to view programmable declarativeness:

- A definitional view, where declarativeness is a property of the component implementation. For example, programs written in the declarative model are guaranteed to be declarative, because of properties of the model.
- An observational view, where declarativeness is a property of the component in-

1. A Turing machine is a simple formal model of computation, first defined by Alan Turing, that is as powerful as any computer that can be built, as far as is known in the current state of computer science. That is, any computation that can be programmed on any computer can also be programmed on a Turing machine.

terface. The observational view follows the principle of abstraction: that to use a component it is enough to know its specification without knowing its implementation. The component just has to behave declaratively, i.e., as if it were independent, stateless, and deterministic, without necessarily being written in a declarative computation model.

This book uses both the definitional and observational views. When we are interested in looking inside a component, we will use the definitional view. When we are interested in how a component behaves, we will use the observational view.

Two styles of definitional declarative programming have become particularly popular: the functional and the logical. In the functional style, we say that a component defined as a mathematical function is declarative. Functional languages such as Haskell and Standard ML follow this approach. In the logical style, we say that a component defined as a logical relation is declarative. Logic languages such as Prolog and Mercury follow this approach. It is harder to formally manipulate functional or logical programs than descriptive programs, but they still follow simple algebraic laws.[2] The declarative model used in this chapter encompasses both functional and logic styles.

The observational view lets us use declarative components in a declarative program even if they are written in a nondeclarative model. For example, a database interface can be a valuable addition to a declarative language. Yet, the implementation of this interface is almost certainly not going to be logical or functional. It suffices that it could have been defined declaratively. Sometimes a declarative component will be written in a functional or logical style, and sometimes it will not be. In later chapters we build declarative components in nondeclarative models. We will not be dogmatic about the matter; we will consider the component to be declarative if it behaves declaratively.

3.1.2 Specification languages

Proponents of declarative programming sometimes claim that it allows dispensing with the implementation, since the specification is all there is. That is, the specification is the program. This is true in a formal sense, but not in a practical sense. Practically, declarative programs are very much like other programs: they require algorithms, data structures, structuring, and reasoning about the order of operations. This is because declarative languages can only use mathematics that can be implemented efficiently. There is a trade-off between expressiveness and efficiency. Declarative programs are usually a lot longer than what a specification could be. So the distinction between specification and implementation still makes sense, even for declarative programs.

It is possible to define a declarative language that is much more expressive than what we use in the book. Such a language is called a specification language. It is

2. For programs that do not use the nondeclarative abilities of these languages.

- The **if** statement.
- The **case** statement.
- Procedure declaration, i.e., the statement ⟨x⟩=⟨v⟩ where ⟨v⟩ is a procedure value.

They allow building statements out of other statements. All these ways of combining statements are deterministic (if their component statements are deterministic, then so are they) and they do not depend on any context.

3.2 Iterative computation

We will now look at how to program in the declarative model. We start by looking at a very simple kind of program, the iterative computation. An iterative computation is a loop whose stack size is bounded by a constant, independent of the number of iterations. This kind of computation is a basic programming tool. There are many ways to write iterative programs. It is not always obvious when a program is iterative. Therefore, we start by giving a general schema that shows how to construct many interesting iterative computations in the declarative model.

3.2.1 A general schema

An important class of iterative computations starts with an initial state S_0 and transforms the state in successive steps until reaching a final state S_{final}:

$$S_0 \rightarrow S_1 \rightarrow \cdots \rightarrow S_{\text{final}}$$

An iterative computation of this class can be written as a general schema:

```
fun {Iterate Si}
   if {IsDone Si} then Si
   else Si+1 in
      Si+1={Transform Si}
      {Iterate Si+1}
   end
end
```

In this schema, the functions `IsDone` and `Transform` are problem-dependent. Let us prove that any program that follows this schema is iterative. We will show that the stack size does not grow when executing `Iterate`. For clarity, we give just the statements on the semantic stack, leaving out the environments and the store:

- Assume the initial semantic stack is [R={Iterate S_0}].
- Assume that {IsDone S_0} returns **false**. Just after executing the **if**, the semantic stack is [S_1={Transform S_0},R={Iterate S_1}].
- After executing {Transform S_0}, the semantic stack is [R={Iterate S_1}].

algorithm therefore always converges.

Figure 3.4 shows one way of defining Newton's method as an iterative computation. The function {`SqrtIter Guess X`} calls {`SqrtIter {Improve Guess X} X`} until `Guess` satisfies the condition {`GoodEnough Guess X`}. It is clear that this is an instance of the general schema, so it is an iterative computation. The improved guess is calculated according to the formula given above. The "good enough" check is $|x - g^2|/x < 0.00001$, i.e., the square root has to be accurate to five decimal places. This check is relative, i.e., the error is divided by x. We could also use an absolute check, e.g., something like $|x - g^2| < 0.00001$, where the magnitude of the error has to be less than some constant. Why is using a relative check better when calculating square roots?

3.2.3 Using local procedures

In the Newton's method program of figure 3.4, several "helper" routines are defined: `SqrtIter`, `Improve`, `GoodEnough`, and `Abs`. These routines are used as building blocks for the main function `Sqrt`. In this section, we discuss where to define helper routines. The basic principle is that a helper routine defined only as an aid to define another routine should not be visible elsewhere. (We use the word "routine" for both functions and procedures.)

In the Newton example, `SqrtIter` is only needed inside `Sqrt`, `Improve` and `GoodEnough` are only needed inside `SqrtIter`, and `Abs` is a utility function that could be used elsewhere. There are two basic ways to express this visibility, with somewhat different semantics. The first way is shown in figure 3.5: the helper routines are defined outside of `Sqrt` in a **`local`** statement. The second way is shown in figure 3.6: each helper routine is defined inside of the routine that needs it.[3]

In figure 3.5, there is a trade-off between readability and visibility: `Improve` and `GoodEnough` could be defined local to `SqrtIter` only. This would result in two levels of local declarations, which is harder to read. We have decided to put all three helper routines in the same local declaration.

In figure 3.6, each helper routine sees the arguments of its enclosing routine as external references. These arguments are precisely those with which the helper routines are called. This means we could simplify the definition by removing these arguments from the helper routines. This gives figure 3.7.

There is a trade-off between putting the helper definitions outside the routine that needs them or putting them inside:

- Putting them inside (figures 3.6 and 3.7) lets them see the arguments of the main routines as external references, according to the lexical scoping rule (see section 2.4.3). Therefore, they need fewer arguments. But each time the main routine is invoked, new helper routines are created. This means that new procedure

3. We leave out the definition of `Abs` to avoid needless repetition.

```
local
   fun {Improve Guess X}
     (Guess + X/Guess) / 2.0
   end
   fun {GoodEnough Guess X}
     {Abs X-Guess*Guess}/X < 0.00001
   end
   fun {SqrtIter Guess X}
      if {GoodEnough Guess X} then Guess
      else
         {SqrtIter {Improve Guess X} X}
      end
   end
in
   fun {Sqrt X}
      Guess=1.0
   in
      {SqrtIter Guess X}
   end
end
```

Figure 3.5: Finding roots using Newton's method (second version).

```
fun {Sqrt X}
   fun {SqrtIter Guess X}
      fun {Improve Guess X}
        (Guess + X/Guess) / 2.0
      end
      fun {GoodEnough Guess X}
        {Abs X-Guess*Guess}/X < 0.00001
      end
   in
      if {GoodEnough Guess X} then Guess
      else
         {SqrtIter {Improve Guess X} X}
      end
   end
   Guess=1.0
in
   {SqrtIter Guess X}
end
```

Figure 3.6: Finding roots using Newton's method (third version).

```
fun {Sqrt X}
   fun {SqrtIter Guess}
      fun {Improve}
        (Guess + X/Guess) / 2.0
      end
      fun {GoodEnough}
        {Abs X-Guess*Guess}/X < 0.00001
      end
   in
      if {GoodEnough} then Guess
      else
         {SqrtIter {Improve}}
      end
   end
   Guess=1.0
in
   {SqrtIter Guess}
end
```

Figure 3.7: Finding roots using Newton's method (fourth version).

values are created.

- Putting them outside (figures 3.4 and 3.5) means that the procedure values are created once and for all, for all calls to the main routine. But then the helper routines need more arguments so that the main routine can pass information to them.

In figure 3.7, new definitions of `Improve` and `GoodEnough` are created on each iteration of `SqrtIter`, whereas `SqrtIter` itself is only created once. This suggests a good trade-off, where `SqrtIter` is local to `Sqrt` and both `Improve` and `GoodEnough` are outside `SqrtIter`. This gives the final definition of figure 3.8, which we consider the best in terms of both efficiency and visibility.

3.2.4 From general schema to control abstraction

The general schema of section 3.2.1 is a programmer aid. It helps the programmer design efficient programs but it is not seen by the computation model. Let us go one step further and provide the general schema as a program component that can be used by other components. We say that the schema becomes a control abstraction, i.e., an abstraction that can be used to provide a desired control flow. Here is the general schema:

```
fun {SumList Xs}
   case Xs
   of nil then 0
   [] X|Xr then X+{SumList Xr}
   end
end
```

Its type is ⟨**fun** {$ ⟨List ⟨Int⟩⟩} : ⟨Int⟩⟩. The input must be a list of integers because `SumList` internally uses the integer 0. The following call

```
{Browse {SumList [1 2 3]}}
```

displays 6. Since `Xs` can be one of two values, namely `nil` or `X|Xr`, it is natural to use a **case** statement. As in the `Nth` example, not using an **else** in the case will raise an exception if the argument is outside the domain of the function. For example:

```
{Browse {SumList 1|foo}}
```

raises an exception because `1|foo` is not a list, and the definition of `SumList` assumes that its input is a list.

3.4.2.3 Naive definitions are often slow

Let us define a function to reverse the elements of a list. Start with a recursive definition of list reversal:

- Reverse of `nil` is `nil`.
- Reverse of `X|Xs` is `Z`, where
  reverse of `Xs` is `Ys`, and
  append `Ys` and `[X]` to get `Z`.

This works because `X` is moved from the front to the back. Following this recursive definition, we can immediately write a function:

```
fun {Reverse Xs}
   case Xs
   of nil then nil
   [] X|Xr then
      {Append {Reverse Xr} [X]}
   end
end
```

Its type is ⟨**fun** {$ ⟨List⟩} : ⟨List⟩⟩. Is this function efficient? To find out, we have to calculate its execution time given an input list of length n. We can do this rigorously with the techniques of section 3.5. But even without these techniques, we can see intuitively what happens. There will be n recursive calls followed by n calls to `Append`. Each `Append` call will have a list of length $n/2$ on average. The total execution time is therefore proportional to $n \cdot n/2$, namely n^2. This is rather slow. We would expect that reversing a list, which is not exactly a complex calculation, would take time proportional to the input length and not to its square.

Concepts, Techniques, and Models of Computer Programming

PETER VAN ROY and SEIF HARIDI

This innovative text presents computer programming as a unified discipline in a way that is both practical and scientifically sound. The book focuses on techniques of lasting value and explains them precisely in terms of a simple abstract machine. The book presents all major programming paradigms in a uniform framework that shows their deep relationships and how and where to use them together.

After an introduction to programming concepts, the book presents both well-known and lesser-known computation models ("programming paradigms"). Each model has its own set of techniques and each is included on the basis of its usefulness in practice. The general models include declarative programming, declarative concurrency, message-passing concurrency, explicit state, object-oriented programming, shared-state concurrency, and relational programming. Specialized models include graphical user interface programming, distributed programming, and constraint programming. Each model is based on its kernel language—a simple core language that consists of a small number of programmer-significant elements. The kernel languages are introduced progressively, adding concepts one by one, thus showing the deep relationships between different models. The kernel languages are defined precisely in terms of a simple abstract machine. Because a wide variety of languages and programming paradigms can be modeled by a small set of closely related kernel languages, this approach allows programmer and student to grasp the underlying unity of programming. The book has many program fragments and exercises, all of which can be run on the Mozart Programming System, an Open Source software package that features an interactive incremental development environment.

Peter Van Roy is Professor in the Department of Computing Science and Engineering at Université catholique de Louvain, at Louvain-la-Neuve, Belgium. Seif Haridi is Professor of Computer Systems in the Department of Microelectronics and Information Technology at the Royal Institute of Technology, Sweden, and Chief Scientific Advisor of the Swedish Institute of Computer Science.

"In almost 20 years since Abelson and Sussman revolutionized the teaching of computer science with their *Structure and Interpretation of Computer Programs*, this is the first book I've seen that focuses on big ideas and multiple paradigms, as *SICP* does, but chooses a very different core model (declarative programming). I wouldn't have made all the choices Van Roy and Haridi have made, but I learned a lot from reading this book, and I hope it gets a wide audience."
—Brian Harvey, Lecturer, Computer Science Division, University of California, Berkeley

"This book follows in the fine tradition of Abelson/Sussman and Kamin's book on interpreters, but goes well beyond them, covering functional and Smalltalk-like languages as well as more advanced concepts in concurrent programming, distributed programming, and some of the finer points of C++ and Java."
—Peter Norvig, Google Inc.

Cover image:
The still unfinished Expiatory Temple of the Sagrada Família in Barcelona is a metaphor for programming. Marrying diversity of style with organic unity, the temple has an equilibrated construction with complex geometries that are in accord with nature's laws. Photograph by Peter Van Roy with the support of Felix Freitag, Leandro Navarro, and Didier Granier.

The MIT Press
Massachusetts Institute of Technology
Cambridge, Massachusetts 02142
http://mitpress.mit.edu
0-262-22069-5