complexity. This gives a solid foundation to the programmer's intuition and the programming techniques built on top of it.

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

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

Comparison with other approaches

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

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

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

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

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

Designing abstractions

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

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

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

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

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

Main features

Pedagogical approach

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

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

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

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

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

Formalism used

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

Panorama of computation models

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

More is not better (or worse), just different

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

The importance of using models together

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

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

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

The limits of single models

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

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

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

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

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

Teaching from the book

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

Role in informatics curriculum

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

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

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

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

Courses

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

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

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

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

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

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

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

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

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

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

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

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

Software

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

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

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

History and acknowledgements

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

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

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

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

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

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

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

What's missing

There are two main topics missing from the book:

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

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

Final comments

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

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

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

Louvain-la-Neuve, Belgium PETER VAN ROY
Kista, Sweden SEIF HARIDI
June 2003

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 it. Forgetting to do this can result in strange errors if older versions of the variables exist. Starting with Chapter 2 and for all succeeding 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 into your `.ozrc` file, so they will be loaded automatically when the system starts up.

- There are a few differences between the ideal implementation of this book and the Mozart system. They are explained on the book's Web site.

Part I

Introduction

Chapter 1

Introduction to Programming Concepts

"There is no royal road to geometry."
– Euclid's reply to Ptolemy, *Euclid* (*c.* 300 BC)

"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 short-cuts for values. That is, they cannot be assigned more than once. But you *can* declare another variable with the same name as a previous one. This means that the old one is no longer accessible. But previous calculations, which used the old variable, are not changed. This is because there are in fact 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 letters or digits. For example, the capital letter “V” 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. Old calculations using the same identifier `V` 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, that is, 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 keyword **declare** says we want to define something new. The keyword **fun** starts a new function. The function is called `Fact` and has one argument `N`. The argument is a local variable, i.e., it is known only inside the function body. Each time we call the function a new variable is declared.

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 done. This expression uses `Fact`, the very function we are defining! This is called *recursion*. It is perfectly normal and no cause for alarm. `Fact` is recursive because the factorial of `N` is simply `N` times the factorial of `N-1`. `Fact` uses the following mathematical definition of factorial:

$$\begin{aligned} 0! &= 1 \\ n! &= n \times (n-1)! \ \text{ if } \ n > 0 \end{aligned}$$

  which is recursive.

Now we can try out the function:

```
{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:

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

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

which leads naturally to the following function:

```
declare
fun {Comb N R}
   {Fact N} div ({Fact R}*{Fact N-R})
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 function `Comb` calls `Fact` three times. It is always possible to use existing functions to help define new functions. This principle is called *functional abstraction* because it uses functions to build abstractions. In this way, large programs are like onions, with layers upon layers of functions calling functions.

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     2     1
1     3     3     1
```

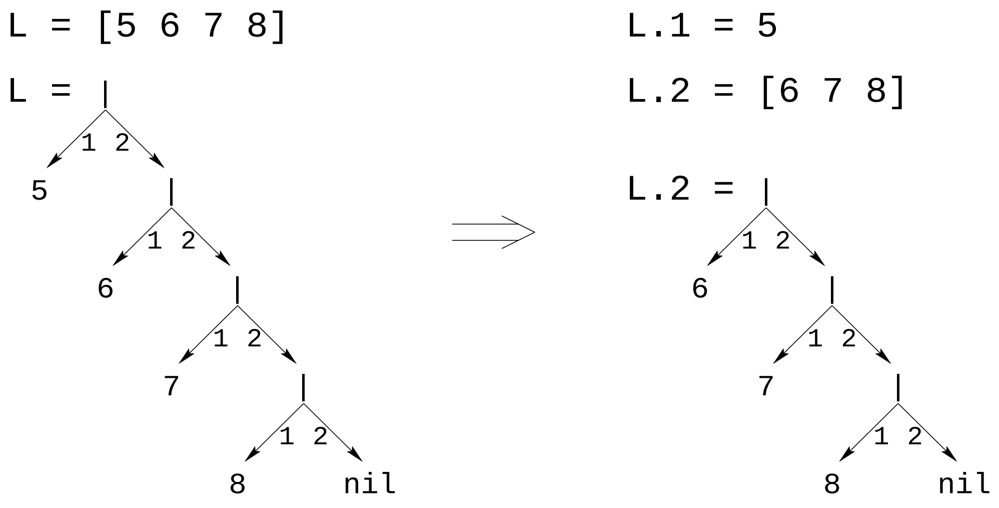

Figure 1.1: Taking apart the list `[5 6 7 8]`

```
      1   4   6   4   1
  .   .   .   .   .   .   .   .   .   .
```

This triangle is named after scientist and mystic Blaise Pascal. It starts with 1 in the first row. Each element is the sum of two other elements: the ones above it and just to the left and right. (If there is no element, like on the edges, then zero is taken.) We would like to define one function that calculates the whole nth row in one swoop. The nth row has n integers in it. We can do it by using *lists* of integers.

A list is just a sequence of elements, bracketed at the left and right, like `[5 6 7 8]`. For historical reasons, the empty list is written `nil` (and not `[]`). Lists can be displayed just like numbers:

```
{Browse [5 6 7 8]}
```

The notation `[5 6 7 8]` is a short-cut. A list is actually a *chain of links*, where each link contains two things: one list element and a reference to the rest of the chain. Lists are always created *one element a time*, starting with `nil` and adding links one by one. A new link is written `H|T`, where `H` is the new element and `T` is the old part of the chain. Let us build a list. We start with `Z=nil`. We add a first link `Y=7|Z` and then a second link `X=6|Y`. Now `X` references a list with two links, a list that can also be written as `[6 7]`.

The link `H|T` is often called a *cons*, a term that comes from Lisp.[1] We also call it a *list pair*. Creating a new link is called *consing*. If `T` is a list, then consing `H` and `T` together makes a new list `H|T`:

[1]Much list terminology was introduced with the Lisp language in the late 1950's and has stuck ever since [120]. Our use of the vertical bar comes from Prolog, a logic programming language that was invented in the early 1970's [40, 182]. Lisp itself writes the cons as `(H . T)`, which it calls a *dotted pair*.

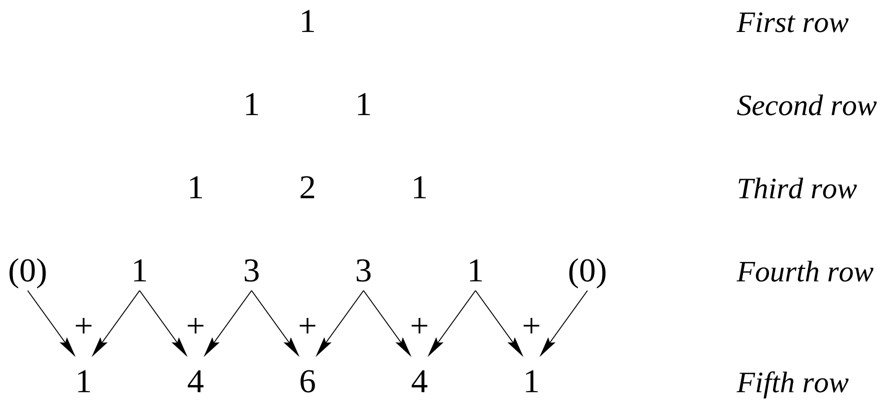

Figure 1.2: Calculating the fifth row of Pascal's triangle

```
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 the `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, that is, between the **`then`** and the **`end`**.

1.5 Functions over lists

Now that we can calculate with lists, let us define a function, `{Pascal N}`, to calculate the nth row of Pascal's triangle. Let us first understand how to do the calculation by hand. Figure 1.2 shows how to calculate the fifth row from the fourth. Let us see how this works if each row is a list of integers. To calculate a row, we start from the previous row. We shift it left by one position and shift it right by one position. We then add the two shifted rows together. For example, take the fourth row:

```
    [1   3   3   1]
```

We shift this row left and right and then add them together:

```
  [1   3   3   1   0]
+ [0   1   3   3   1]
```

Note that shifting left adds a zero to the right and shifting right adds a zero to the left. Doing the addition gives:

```
  [1   4   6   4   1]
```

which is the fifth row.

The main function

Now that we understand how to solve the problem, we can write a function to do the same operations. Here it is:

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

In addition to defining `Pascal`, we declare the variables for the three auxiliary functions that remain to be defined.

The auxiliary functions

This does not completely solve the problem. We have to define three more functions: `ShiftLeft`, which shifts left by one position, `ShiftRight`, which shifts right by one position, and `AddList`, which adds two lists. Here are `ShiftLeft` and `ShiftRight`:

```
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 that 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 you 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 technique we used to write `Pascal`:

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

The technique of first writing the main function and filling in the blanks afterwards is known as *top-down* software development. It is one of the most well-known approaches, but it gives only part of the story.

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 *semantics* of the language.
- 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 task: it means verifying the compiler, the run-time system, the operating system, and the hardware! This is an important topic, but it is beyond the scope of the present book. For this book, we place our trust in the Mozart developers, software companies, and hardware manufacturers.[2]

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 cases. Then we show that, if the program is correct for a given case, then it is correct for the next case. From these two steps, mathematical induction lets us conclude that the program is always correct. This technique can be applied for integers and lists:

- For integers, the base case is 0 or 1, and for a given integer n the next case is $n + 1$.

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

- For lists, the base case is `nil` (the empty list) or a list with one or a few elements, 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.
- Assume that `{Fact N-1}` is correct. Then look at the call `{Fact N}`. We see that the **`if`** instruction takes the **`else`** case, 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. 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, so 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 variable exists only 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 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 processes a list of average size $n/2$. Therefore its time complexity is 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. Another way to evaluate functions is called *lazy* evaluation.[3] In lazy evaluation, a calculation is done only when the result is needed. 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 it is an infinite loop, but it is not. The `lazy` annotation ensures that the function

[3]These are sometimes called *data-driven* and *demand-driven* evaluation, respectively.

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:

```
L={Ints 0}
{Browse L}
```

This displays the following, i.e., nothing at all:

```
L<Future>
```

(The browser displays values but does not affect their calculation.) The "`Future`" annotation means that `L` has a lazy function attached to it. If the value of `L` is needed, then this function will be automatically called. Therefore to get more results, we have to do something that needs the list. For example:

```
{Browse L.1}
```

This displays the first element, namely 0. We can calculate with the list as if it were completely there:

```
case L of A|B|C|_ then {Browse A+B+C} end
```

This causes the first three elements of `L` to be calculated, and no more. What does it display?

Lazy calculation of Pascal's triangle

Let us do something useful with lazy evaluation. We would like to write a function that calculates as many rows of Pascal's triangle as are needed, but we do not know beforehand how many. That is, we have to look at the rows to decide when there are enough. Here is a lazy function that generates an infinite list of rows:

```
fun lazy {PascalList Row}
   Row|{PascalList
          {AddList {ShiftLeft Row}
                   {ShiftRight Row}}}
end
```

Calling this function and browsing it will display nothing:

```
declare
L={PascalList [1]}
{Browse L}
```

(The argument `[1]` is the first row of the triangle.) To display more results, they have to be needed:

```
{Browse L.1}
{Browse L.2.1}
```

This displays the first and second rows.

Instead of writing a lazy function, we could write a function that takes `N`, the number of rows we need, and directly calculates those rows starting from an initial row:

```
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. Can we somehow just have *one* generic version? 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 an addition `H1+H2`, this version does `{Op H1 H2}`.

Variations on Pascal's triangle

Let us define some functions to try out `GenericPascal`. To get the original Pascal's triangle, we can define the addition function:

```
fun {Add X Y} X+Y end
```

Now we can run `{GenericPascal Add 5}`.[4] This gives the fifth row exactly as before. We can define `FastPascal` using `GenericPascal`:

```
fun {FastPascal N} {GenericPascal Add N} end
```

Let us define another function:

```
fun {Xor X Y} if X==Y then 0 else 1 end end
```

This does an *exclusive-or* operation, which is defined as follows:

X	Y	{Xor X Y}
0	0	0
0	1	1
1	0	1
1	1	0

Exclusive-or lets us calculate the *parity* of each number in Pascal's triangle, i.e., whether the number is odd or even. The numbers themselves are not calculated. Calling `{GenericPascal Xor N}` gives the result:

```
                  1
               1     1
            1     0     1
         1     1     1     1
      1     0     0     0     1
   1     1     0     0     1     1
1     0     1     0     1     0     1
. . . . . . . . . . . . . .
```

Some other functions are given in the exercises.

1.10 Concurrency

We would like our program to have several independent activities, each of which executes at its own pace. This is called *concurrency*. There should be no interference between the activities, unless the programmer decides that they need to

[4]We can also call `{GenericPascal Number.´+´ 5}`, since the addition operation `´+´` is part of the module `Number`. But modules are not introduced in this chapter.

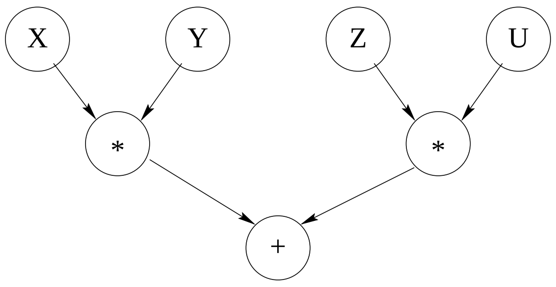

Figure 1.3: A simple example of dataflow execution

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 for example:

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