Texts in Computing

Volume 4

The Haskell Road to Logic, Maths and Programming

second edition

Volume 1
Programming Languages and Operational Semantics
Maribel Fernández

Volume 2
An Introduction to Lambda Calculi for Computer Scientists
Chris Hankin

Volume 3
Logical Reasoning: A First Course
Rob Nederpelt and Fairouz Kamareddine

Volume 4
The Haskell Road to Logic, Maths and Programming
Kees Doets and Jan van Eijck

Texts in Computing Series Editor
Ian Mackie, King's College London

The Haskell Road to Logic, Maths and Programming

second edition

Kees Doets

ILLC, Amsterdam

Jan van Eijck

CWI, Amsterdam ILLC, Amsterdam

ISBN 0-9543006-9-6
King's College Publications
Scientific Director: Dov Gabbay
Managing Director: Jane Spurr
Department of Computer Science
Strand, London WC2R 2LS, UK
kcp@dcs.kcl.ac.uk

Cover design by Richard Fraser, www.avalonarts.co.uk
Printed by Lightning Source, Milton Keynes, UK

Contents

Preface

Purpose

Long ago, when Alexander the Great asked the mathematician Menaechmus for a crash course in geometry, he got the famous reply "There is no royal road to mathematics." Where there was no shortcut for Alexander, there is no shortcut for us. Still, the fact that we have access to computers and mature programming languages means that there are avenues for us that were denied to the kings and emperors of yore.

The purpose of this book is to teach logic and mathematical reasoning in practice, and to connect logical reasoning with computer programming. The programming language that will be our tool for this is Haskell, a member of the Lisp family. Haskell emerged in the last decade as a standard for lazy functional programming, a programming style where arguments are evaluated only when the value is actually needed. Functional programming is a form of descriptive programming, very different from the style of programming that you find in prescriptive languages like C or Java. Haskell is based on a logical theory of computable functions called the lambda calculus.

> Lambda calculus is a formal language capable of expressing arbitrary computable functions. In combination with types it forms a compact way to denote on the one hand functional programs and on the other hand mathematical proofs. [Bar84]

Haskell can be viewed as a particularly elegant implementation of the lambda calculus. It is a marvelous demonstration tool for logic and maths because its functional character allows implementations to remain very close to the concepts that get implemented, while the laziness permits smooth handling of infinite data structures.

Haskell syntax is easy to learn, and Haskell programs are constructed and tested in a modular fashion. This makes the language well suited for fast prototyping. Programmers find to their surprise that implementation

of a well-understood algorithm in Haskell usually takes far less time than implementation of the same algorithm in other programming languages. Getting familiar with new algorithms through Haskell is also quite easy. Learning to program in Haskell is learning an extremely useful skill.

Throughout the text, abstract concepts are linked to concrete representations in Haskell. Haskell comes with an easy to use interpreter, Hugs. Haskell compilers, interpreters and documentation are freely available from the Internet [HT]. Everything one has to know about programming in Haskell to understand the programs in the book is explained as we go along, but we do not cover every aspect of the language. For a further introduction to Haskell we refer the reader to [HFP96].

Logic in Practice

The subject of this book is the *use* of logic in practice, more in particular the use of logic in reasoning about programming tasks. Logic is not taught here as a mathematical discipline per se, but as an aid in the understanding and construction of proofs, and as a tool for reasoning about formal objects like numbers, lists, trees, formulas, and so on. As we go along, we will introduce the concepts and tools that form the set-theoretic basis of mathematics, and demonstrate the role of these concepts and tools in implementations. These implementations can be thought of as *representations* of the mathematical concepts.

Although it may be argued that the logic that is needed for a proper understanding of reasoning in reasoned programming will get acquired more or less automatically in the process of learning (applied) mathematics and/or programming, students nowadays enter university without any experience whatsoever with mathematical proof, the central notion of mathematics.

The rules of Chapter 3 represent a detailed account of the structure of a proof. The purpose of this account is to get the student acquainted with proofs by putting emphasis on logical structure. The student is encouraged to write "detailed" proofs, with every logical move spelled out in full. The next goal is to move on to writing "concise" proofs, in the customary mathematical style, while keeping the logical structure in mind. Once the student has arrived at this stage, most of the logic that is explained in Chapter 3 can safely be forgotten, or better, can safely fade into the subconsciousness of the matured mathematical mind.

Pre- and Postconditions of Use

We do not assume that our readers have previous experience with either programming or construction of formal proofs. We do assume previous acquaintance with mathematical notation, at the level of secondary school mathematics. Wherever necessary, we will recall relevant facts. Everything one needs to know about mathematical reasoning or programming is explained as we go along. We do assume that our readers are able to retrieve software from the Internet and install it, and that they know how to use an editor for constructing program texts.

After having worked through the material in the book, i.e., after having digested the text and having carried out a substantial number of the exercises, the reader will be able to write interesting programs, reason about their correctness, and document them in a clear fashion. The reader will also have learned how to set up mathematical proofs in a structured way, and how to read and digest mathematical proofs written by others.

How to Use the Book

Chapters 1–7 of the book are devoted to a gradual introduction of the concepts, tools and methods of mathematical reasoning and reasoned programming.

Chapter 8 tells the story of how the various number systems (natural numbers, integers, rationals, reals, complex numbers) can be thought of as constructed in stages from the natural numbers. Everything gets linked to the implementations of the various Haskell types for numerical computation.

Chapter 9 starts with the question of how to automate the task of finding closed forms for polynomial sequences. It is demonstrated how this task can be automated with difference analysis plus Gaussian elimination. Next, polynomials are implemented as lists of their coefficients, with the appropriate numerical operations, and it is shown how this representation can be used for solving combinatorial problems.

Chapter 10 provides the first general textbook treatment (as far as we know) of the important topic of corecursion. The chapter presents the proof methods suitable for reasoning about corecursive data types like streams and processes, and then goes on to introduce power series as infinite lists of coefficients, and to demonstrate the uses of this representation for handling combinatorial problems. This generalizes the use of polynomials for combinatorics.

Chapter 11 offers a guided tour through Cantor's paradise of the infinite, while providing extra challenges in the form of a wide range of additional

exercises.

The book can be used as a course textbook, but since it comes with solutions to all exercises (electronically available from the authors upon request) it is also well suited for private study. Courses based on the book could start with Chapters 1–7, and then make a choice from the remaining Chapters. Here are some examples:

Road to Numerical Computation Chapters 1–7, followed by 8 and 9.

Road to Streams and Corecursion Chapters 1–7, followed by 9 and 10.

Road to Cantor's Paradise Chapters 1–7, followed by 11.

Study of the remaining parts of the book can then be set as individual tasks for students ready for an extra challenge. The guidelines for setting up formal proofs in Chapter 3 should be recalled from time to time while studying the book, for proper digestion.

Exercises

Parts of the text and exercises marked by a * are somewhat harder than the rest of the book. All exercises are solved in the electronically available solutions volume. Before turning to these solutions, one should read the *Important Advice to the Reader* that this volume starts with.

Book Website and Contact

The programs in this book have all been tested with Hugs98, the version of Hugs that implements the Haskell 98 standard. The full source code of all programs is integrated in the book; in fact, each chapter can be viewed as a *literate program* [Knu92] in Haskell. The source code of all programs discussed in the text can be found on the website devoted to this book, at address `http://www.cwi.nl/~jve/HR`. Here you can also find a list of errata, and further relevant material.

Readers who want to share their comments with the authors are encouraged to get in touch with us at email address `jve@cwi.nl`.

Acknowledgments

Remarks from the people listed below have sparked off numerous improvements. Thanks to Johan van Benthem, Jan Bergstra, Jacob

Brunekreef, Thierry Coquand (who found the lecture notes on the internet and sent us his comments), Tim van Erven, Wan Fokkink, Evan Goris, Robbert de Haan, Sandor Heman, Eva Hoogland, Rosalie Iemhoff, Dick de Jongh, Anne Kaldewaij, Breanndán Ó Nualláin, Alban Ponse, Vincent van Oostrom, Piet Rodenburg, Jan Rutten, Marco Swaen, Jan Terlouw, John Tromp, Yde Venema, Albert Visser and Stephanie Wehner for suggestions and criticisms. The beautiful implementation of the sieve of Eratosthenes in Section 3.7 was suggested to us by Fer-Jan de Vries. Jack Jansen, Hayco de Jong and Jurgen Vinju provided instant support with running Haskell on non-Linux machines.

The course on which this book is based was developed at ILLC (the Institute of Logic, Language and Computation of the University of Amsterdam) with financial support from the *Spinoza Logic in Action* initiative of Johan van Benthem, which is herewith gratefully acknowledged. We also wish to thank ILLC and CWI (Centrum voor Wiskunde en Informatica, or Centre for Mathematics and Computer Science, also in Amsterdam), the home institute of the second author, for providing us with a supportive working environment. CWI has kindly granted permission to reuse material from [Doe96].

It was Krzysztof Apt who, perceiving the need of a deadline, spurred us on to get in touch with a publisher and put ourselves under contract. Finally, many thanks go to Ian Mackie, Texts in Computing Series Editor, of King's College, London, for swift decision making, prompt and helpful feedback and enthousiastic support during the preparation of the manuscript for the *Text in Computing* series.

Additions to the Second Edition

Ralf Laemmel suggested the use of Haskell type classes to give a general treatment of propositional logic, in Chapter 2.

The chapter on Polynomials (Chapter 9) was expanded with a new section on computing polynomial representations for number sequences using the calculus of finite differences.

The Haskell code of the chapters was brought up to date to comply with the new Haskell 2010 standard. Aaron Denney and Ralf Laemmel helped to bring `Nats.hs` in conformance with the Haskell standard.

Corrections

In this second edition a number of errata from the first edition were corrected, based on error reports we received from: Aaron Denney, Max Eckenbach, Ralf Laemmel, Brian Lewis, Simon Mayo, Torsten Marek, Carl Meijer, Wiesław Poszewiecki, Shyamal Prasad, John Rood, Philipp Schneider, J.A.Zaratiegui. Tijs van der Storm gave advice on typography. Thank you all very much.

Chapter 1

Getting Started

Preview

Our purpose is to teach logic and mathematical reasoning in practice, and to connect formal reasoning to computer programming. It is convenient to choose a programming language for this that permits implementations to remain as close as possible to the formal definitions. Such a language is the functional programming language Haskell [HT]. Haskell was named after the logician Haskell B. Curry. Curry, together with Alonzo Church, laid the foundations of functional computation in the era Before the Computer, around 1940. As a functional programming language, Haskell is a member of the Lisp family. Others family members are Scheme, ML, Occam, Clean. Haskell98 is intended as a standard for lazy functional programming. Lazy functional programming is a programming style where arguments are evaluated only when the value is actually needed.

With Haskell, the step from formal definition to program is particularly easy. This presupposes, of course, that you are at ease with formal definitions. Our reason for combining training in reasoning with an introduction to functional programming is that your programming needs will provide motivation for improving your reasoning skills. Haskell programs will be used as illustrations for the theory throughout the book. We will always put computer programs and pseudo-code of algorithms in frames (rectangular boxes).

The chapters of this book are written in so-called 'literate programming' style [Knu92]. Literate programming is a programming style where the program and its documentation are generated from the same source. The text of every chapter in this book can be viewed as the documentation of

the program code in that chapter. Literate programming makes it impossible for program and documentation to get out of sync. Program documentation is an integrated part of literate programming, in fact the bulk of a literate program is the program documentation. When writing programs in literate style there is less temptation to write program code first while leaving the documentation for later. Programming in literate style proceeds from the assumption that the main challenge when programming is to make your program digestible for humans. For a program to be useful, it should be easy for others to understand the code. It should also be easy for you to understand your own code when you reread your stuff the next day or the next week or the next month and try to figure out what you were up to when you wrote your program.

To save you the trouble of retyping, the code discussed in this book can be retrieved from the book website. The program code is the text in `typewriter` font that you find in rectangular boxes throughout the chapters. Boxes may also contain code that is not included in the chapter modules, usually because it defines functions that are already predefined by the Haskell system, or because it redefines a function that is already defined elsewhere in the chapter.

Typewriter font is also used for pieces of interaction with the Haskell interpreter, but these illustrations of how the interpreter behaves when particular files are loaded and commands are given are not boxed.

Every chapter of this book is a so-called Haskell module. The following two lines declare the Haskell module for the Haskell code of the present chapter. This module is called `GS`.

```
module GS

where
```

1.1 Starting up the Haskell Interpreter

We assume that you succeeded in downloading and installing the Haskell platform (`http://hackage.haskell.org/platform/`). This includes the Haskell interpreter *ghci*. You can start the interpreter by typing `ghci` at the system prompt. When you start *ghci* you should see something like Figure 1.1.

```
jve@vuur:~/books/hrne$ ghci
GHCi, version 7.4.1: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude>
```

Figure 1.1: Starting up the Haskell interpreter Ghci.

Alternatively, you can use the Haskell interpreter *hugs* (available from `http://www.haskell.org/hugs/`). When you start *hugs* you should see something like Figure 1.2.

```
jve@vuur:~/books/hrne$ hugs
__   __ __  __  ____   ___       ------------------------------------------
||   || ||  || ||  || ||__       Hugs 98: Based on the Haskell 98 standard
||___|| ||__|| ||__||  __||      Copyright (c) 1994-2005
||---||           ___||           World Wide Web: http://haskell.org/hugs
||   ||                           Bugs: http://hackage.haskell.org/trac/hugs
||   || Version: September 2006 ------------------------------------------

Haskell 98 mode: Restart with command line option -98 to enable extensions

Type :? for help
Hugs>
```

Figure 1.2: Starting up the Haskell interpreter Hugs.

The string `Prelude>` on the last line is the Haskell prompt when no user-defined files are loaded.

You can use the Haskell interpreter as a calculator as follows:

```
Prelude> 2^16
65536
Prelude>
```

The string `Prelude>` is the system prompt. `2^16` is what you type. After you hit the return key (the key that is often labeled with *Enter* or ↩), the system answers `65536` and the prompt `Prelude>` reappears.

Exercise 1.1 Try out a few calculations using `*` for multiplication, `+` for addition, `-` for subtraction, `^` for exponentiation, `/` for division. By playing with the system, find out what the precedence order is among these operators.

Parentheses can be used to override the built-in operator precedences:

```
Prelude> (2 + 3)^4
625
```

To quit the interpreter, type `:quit` or `:q` at the system prompt.

1.2 Implementing a Prime Number Test

Suppose we want to implement a definition of *prime number* in a procedure that recognizes prime numbers. A prime number is a natural number greater than 1 that has no proper divisors other than 1 and itself. The natural numbers are $0,1,2,3,4,\ldots$ The list of prime numbers starts with $2,3,5,7,11,13,\ldots$ Except for 2, all of these are odd, of course.

Let $n > 1$ be a natural number. Then we use $\mathrm{LD}(n)$ for the least natural number greater than 1 that divides n. A number d divides n if there is a natural number a with $a \cdot d = n$. In other words, d divides n if there is a natural number a with $\frac{n}{d} = a$, i.e., division of n by d leaves no remainder. Note that $\mathrm{LD}(n)$ exists for every natural number $n > 1$, for the natural number $d = n$ is greater than 1 and divides n. Therefore, the set of divisors of n that are greater than 1 is non-empty. Thus, the set will have a least element.

The following proposition gives us all we need for implementing our prime number test:

Proposition 1.2

1. If $n > 1$ then $\mathrm{LD}(n)$ is a prime number.
2. If $n > 1$ and n is not a prime number, then $(\mathrm{LD}(n))^2 \leqslant n$.

In the course of this book you will learn how to prove propositions like this.

Here is the proof of the first item. This is a proof by contradiction (see Chapter 3). Suppose, for a contradiction that $c = \mathrm{LD}(n)$ is not a prime. Then there are natural numbers a and b with $c = a \cdot b$, and also $1 < a$ and $a < c$. But then a divides n, and contradiction with the fact that c is the smallest natural number greater than 1 that divides n. Thus, $\mathrm{LD}(n)$ must be a prime number.

For a proof of the second item, suppose that $n > 1$, n is not a prime and that $p = \mathrm{LD}(n)$. Then there is a natural number $a > 1$ with $n = p \cdot a$. Thus, a divides n. Since p is the smallest divisor of n with $p > 1$, we have that $p \leqslant a$, and therefore $p^2 \leqslant p \cdot a = n$, i.e., $(\mathrm{LD}(n))^2 \leqslant n$. ∎

The operator $\cdot$ in $a \cdot b$ is a so-called *infix* operator. The operator is written *between* its formal arguments. If an operator is written *before* its formal arguments we call this *prefix* notation. The product of a and b in prefix notation would look like this: $\cdot\ a\ b$.

In writing functional programs, the standard is prefix notation. In an expression `op a b`, `op` is the *function*, and `a` and `b` are the *formal arguments*. The convention is that function application associates to the left, so the expression `op a b` is interpreted as `(op a) b`.

Using prefix notation, we define the operation `divides` that takes two integer expressions and produces a *truth value*. The truth values *true* and *false* are rendered in Haskell as `True` and `False`, respectively.

The integer expressions that the procedure needs to work with are called the *formal arguments* of the procedure. The truth value that it produces is called the *value* of the procedure.

Obviously, m divides n if and only if the remainder of the process of dividing n by m equals 0. The definition of `divides` can therefore be phrased in terms of a predefined procedure `rem` for finding the remainder of a division process:

```
divides d n = rem n d == 0
```

The definition illustrates that Haskell uses `=` for 'is defined as' and `==` for identity. (The Haskell symbol for non-identity is `/=`.)

A line of Haskell code of the form `foo t = ...` (or `foo t1 t2 = ...`, or `foo t1 t2 t3 = ...`, and so on) is called a Haskell *equation*. In such an equation, `foo` is called the *function*, and `t` its *formal argument*.

Thus, in the Haskell equation `divides d n = rem n d == 0`, `divides` is the function, `d` is the first formal argument, and `n` is the second formal argument.

Exercise 1.3 Put the definition of `divides` in a file *prime.hs*. Start the Haskell interpreter (Section 1.1). Now give the command `:load prime` or `:l prime`, followed by pressing *Enter*. Note that `l` is the letter *l*, not the digit 1. (Next to `:l`, a very useful command after you have edited a file of Haskell code is `:reload` or `:r`, for reloading the file.)

```
Prelude> :l prime
Main>
```

The string `Main>` is the Haskell prompt indicating that user-defined files are loaded. This is a sign that the definition was added to the system. The newly defined operation can now be executed, as follows:

```
Main> divides 5 7
False
Main>
```

What appears after the Haskell prompt `Main>` on the first line is what you type. When you press *Enter* the system answers with the second line, followed by the Haskell prompt. You can then continue with:

```
Main> divides 5 30
True
```

It is clear from the proposition above that all we have to do to implement a primality test is to give an implementation of the function LD. It is convenient to define LD in terms of a second function LDF, for the least divisor starting from a given threshold k, with $k \leqslant n$. Thus, $\mathrm{LDF}(k)(n)$ is the least divisor of n, provided n has no divisors $< k$. Clearly, $\mathrm{LD}(n) = \mathrm{LDF}(2)(n)$. Now we can implement LD as follows:

```
ld n = ldf 2 n
```

This leaves the implementation `ldf` of LDF (details of the coding will be explained below):

```
ldf k n | divides k n = k
        | k^2 >  n    = n
        | otherwise   = ldf (k+1) n
```

The definition employs the Haskell operation `^` for exponentiation, `>` for 'greater than', and `+` for addition.

The definition of `ldf` makes use of *equation guarding*. The first line of the `ldf` definition handles the case where the first argument divides the second argument. Every next line assumes that the previous lines do not

apply. The second line handles the case where the first argument does not divide the second argument, and the square of the first argument is greater than the second argument. The third line assumes that the first and second cases do not apply and handles all other cases, i.e., the cases where k does not divide n and $k^2 < n$.

The definition employs the Haskell condition operator `|` . A Haskell equation of the form

```
foo t | condition = ...
```

is called a *guarded equation*. We might have written the definition of `ldf` as a list of guarded equations, as follows:

```
ldf k n | divides k n = k
ldf k n | k^2 >  n    = n
ldf k n               = ldf (k+1) n
```

The expression `condition`, of type `Bool` (i.e., Boolean or truth value), is called the *guard* of the equation.

A list of guarded equations such as

```
foo t | condition_1 = body_1
foo t | condition_2 = body_2
foo t | condition_3 = body_3
foo t               = body_4
```

can be abbreviated as

```
foo t | condition_1 = body_1
      | condition_2 = body_2
      | condition_3 = body_3
      | otherwise   = body_4
```

Such a Haskell definition is read as follows:

- in case `condition_1` holds, `foo t` is by definition equal to `body_1`,
- in case `condition_1` does not hold but `condition_2` holds, `foo t` is by definition equal to `body_2`,
- in case `condition_1` and `condition_2` do not hold but `condition_3` holds, `foo t` is by definition equal to `body_3`,
- and in case none of `condition_1`, `condition_2` and `condition_3` hold, `foo t` is by definition equal to `body_4`.

When we are at the end of the list we know that none of the cases above in the list apply. This is indicated by means of the Boolean constant `otherwise`, defined in the Prelude as `True`. The use of `otherwise` helps to make guards more readable.

Note that the procedure `ldf` is called again from the body of its own definition. We will encounter such **recursive** procedure definitions again and again in the course of this book (see in particular Chapter 7).

Exercise 1.4 Suppose in the definition of `ldf` we replace the condition `k^2 > n` by `k^2 >= n`, where `>=` expresses 'greater than or equal'. Would that make any difference to the meaning of the program? Why (not)?

Now we are ready for a definition of `prime0`, our first implementation of the test for being a prime number.

```
prime0 n | n < 1     = error "not a positive integer"
         | n == 1    = False
         | otherwise = ld n == n
```

Haskell allows a call to the `error` operation in any definition. This is used to break off operation and issue an appropriate message when the primality test is used for numbers below 1. Note that `error` has a parameter of type `String` (indicated by the double quotes). The definition employs the Haskell operation `<` for 'less than'.

Intuitively, what the definition `prime0` says is this:

1. the primality test should not be applied to numbers below 1,
2. if the test is applied to the number 1 it yields 'false',
3. if it is applied to an integer n greater than 1 it boils down to checking that $\mathrm{LD}(n) = n$. In view of the proposition we proved above, this is indeed a correct primality test.

Exercise 1.5 Add these definitions to the file *prime.hs* and try them out.

Remark. The use of variables in functional programming has much in common with the use of variables in logic. The definition `divides d n = rem n d == 0` is equivalent to `divides x y = rem y x == 0`. This is because the variables denote *arbitrary* elements of the type over which they range. They behave like universally quantified variables, and just as in logic the definition does not depend on the variable names. ∎

1.3 Haskell Type Declarations

Haskell has a concise way to indicate that `divides` consumes an integer, then another integer, and produces a truth value (called `Bool` in Haskell). Integers and truth values are examples of *types*. See Section 2.1 for more on the type `Bool`. Section 1.6 gives more information about types in general. Arbitrary precision integers in Haskell have type `Integer`. The following line gives a so-called *type declaration* for the `divides` function.

```
divides :: Integer -> Integer -> Bool
```

`Integer -> Integer -> Bool` is short for `Integer -> (Integer -> Bool)`. A type of the form `a -> b` classifies procedures that take an argument of type `a` to produce a result of type `b`. Thus, `divides` takes an argument of type `Integer` and produces a result of type `Integer -> Bool`, i.e., a procedure that takes an argument of type `Integer`, and produces a result of type `Bool`.

The full code for `divides`, including the type declaration, looks like this:

```
divides :: Integer -> Integer -> Bool
divides d n = rem n d == 0
```

If `d` is an expression of type `Integer`, then `divides d` is an expression of type `Integer -> Bool`. The shorthand that we will use for

`d` *is an expression of type* `Integer`

is: `d :: Integer`.

Exercise 1.6 Can you gather from the definition of `divides` what the type declaration for `rem` would look like?

Exercise 1.7 The Haskell system has a command for checking the types of expressions. Can you explain the following (please try it out; make sure that the file with the definition of `divides` is loaded, together with the type declaration for `divides`):

```
Main> :t divides 5
divides 5 :: Integer -> Bool
Main> :t divides 5 7
divides 5 7 :: Bool
Main>
```

The expression `divides 5 :: Integer -> Bool` is called a *type judgment*. Type judgments in Haskell have the form `expression :: type`.

In Haskell it is not strictly necessary to give explicit type declarations. For instance, the definition of `divides` works quite well without the type declaration, since the system can infer the type from the definition. However, it is good programming practice to give explicit type declarations even when this is not strictly necessary. These type declarations are an aid to understanding, and they greatly improve the digestibility of functional programs for human readers. A further advantage of the explicit type declarations is that they facilitate detection of programming mistakes on the basis of type errors generated by the interpreter. You will find that many programming errors already come to light when your program gets loaded. The fact that your program is well typed does not entail that it is correct, of course, but many incorrect programs do have typing mistakes.

The full code for `ld`, including the type declaration, looks like this:

```
ld :: Integer -> Integer
ld n = ldf 2 n
```

The full code for `ldf`, including the type declaration, looks like this:

```
ldf :: Integer -> Integer -> Integer
ldf k n | divides k n = k
        | k^2 >  n    = n
        | otherwise   = ldf (k+1) n
```

The first line of the code states that the operation `ldf` takes two integers and produces an integer.

The full code for `prime0`, including the type declaration, runs like this:

```
prime0 :: Integer -> Bool
prime0 n | n < 1     = error "not a positive integer"
         | n == 1    = False
         | otherwise = ld n == n
```

The first line of the code declares that the operation `prime0` takes an integer and produces (or *returns*, as programmers like to say) a Boolean (truth value).

In programming generally, it is useful to keep close track of the nature of the objects that are being represented. This is because representations have to be stored in computer memory, and one has to know how much space to allocate for this storage. Still, there is no need to always specify the nature of each data-type explicitly. It turns out that much information about the nature of an object can be inferred from how the object is handled in a particular program, or in other words, from the *operations* that are performed on that object.

Take again the definition of `divides`. It is clear from the definition that an operation is defined with two formal arguments, both of which are of a type for which `rem` is defined, and with a result of type `Bool` (for `rem n d == 0` is a statement that can turn out true or false). If we check the type of the built-in procedure `rem` we get:

```
Prelude> :t rem
rem :: Integral a => a -> a -> a
```

In this particular case, the type judgment gives a *type scheme* rather than a type. It means: if `a` is a type of class `Integral`, then `rem` is of type `a -> a -> a`. Here `a` is used as a variable ranging over types.

In Haskell, `Integral` is the class (see Section 4.2) consisting of the two types for integer numbers, `Int` and `Integer`. The difference between `Int` and `Integer` is that objects of type `Int` have fixed precision, objects of type `Integer` have arbitrary precision.

The type of `divides` can now be inferred from the definition. This is what we get when we load the definition of `divides` without the type declaration:

```
Main> :t divides
divides :: Integral a => a -> a -> Bool
```

1.4 Identifiers in Haskell

In Haskell, there are two kinds of identifiers:

- Variable identifiers are used to name functions. They have to start with a lower-case letter. E.g., `map`, `max`, `fct2list`, `fctToList`, `fct_to_list`.

- Constructor identifiers are used to name types. They have to start with an upper-case letter. Examples are `True`, `False`.

Functions are operations on data-structures, constructors are the building blocks of the data structures themselves (trees, lists, Booleans, and so on).

Names of functions always start with lower-case letters, and may contain both upper- and lower-case letters, but also digits, underscores and the prime symbol `'`. The following *reserved keywords* have special meanings and cannot be used to name functions.

case	class	data	default	deriving	do	else
if	import	in	infix	infixl	infixr	instance
let	module	newtype	of	then	type	where
_						

The use of these keywords will be explained as we encounter them. `_` at the beginning of a word is treated as a lower-case character. The underscore character `_` all by itself is a reserved word for the wildcard pattern that matches anything (page 137).

There is one more reserved keyword that is particular to Hugs: *forall*, for the definition of functions that take polymorphic arguments. See the Hugs documentation for further particulars.

1.5 Playing the Haskell Game

This section consists of a number of further examples and exercises to get you acquainted with the programming language of this book. To save you the trouble of keying in the programs below, you should retrieve the module `GS.hs` for the present chapter from the book website and load it in the interpreter. This will give you a system prompt `GS>`, indicating that all the programs from this chapter are loaded.

In the next example, we use `Int` for the type of fixed precision integers, and `[Int]` for lists of fixed precision integers.

Example 1.8 Here is a function that gives the minimum of a list of integers:

```
mnmInt :: [Int] -> Int
mnmInt [] = error "empty list"
mnmInt [x] = x
mnmInt (x:xs) = min x (mnmInt xs)
```

This uses the predefined function `min` for the minimum of two integers. It also uses pattern matching for lists . The list pattern `[]` matches only the empty list, the list pattern `[x]` matches any singleton list, the list pattern `(x:xs)` matches any non-empty list. A further subtlety is that pattern matching in Haskell is sensitive to order. If the pattern `[x]` is found before `(x:xs)` then `(x:xs)` matches any non-empty list that is not a unit list. See Section 4.6 for more information on list pattern matching.

It is common Haskell practice to refer to non-empty lists as `x:xs`, `y:ys`, and so on, as a useful reminder of the facts that `x` is an element of a list of `x`'s and that `xs` is a list.

Here is a home-made version of `min`:

```
min' :: Int -> Int -> Int
min' x y | x <= y    = x
         | otherwise = y
```

You will have guessed that `<=` is Haskell code for $\leqslant$.

Objects of type `Int` are fixed precision integers. Their range can be found with:

```
Prelude> minBound :: Int
-9223372036854775808
Prelude> maxBound :: Int
9223372036854775807
```

This was *ghci*. With *hugs*, we get different values:

```
Hugs> minBound :: Int
-2147483648
Hugs> maxBound :: Int
2147483647
```

Since $2147483647 = 2^{31} - 1$, we can conclude that the Hugs implementation uses four bytes (32 bits) to represent objects of this type. The Ghci interpreter uses eight bytes (64 bits): $9223372036854775807 = 2^{63} - 1$.

`Integer` is for arbitrary precision integers: the storage space that gets allocated for `Integer` objects depends on the size of the object.

Exercise 1.9 Define a function that gives the maximum of a list of integers. Use the predefined function `max`.

Conversion from Prefix to Infix in Haskell A function can be converted to an infix operator by putting its name in back quotes, like this:

```
Prelude> max 4 5
5
Prelude> 4 `max` 5
5
```

Conversely, an infix operator is converted to prefix by putting the operator in round brackets (p. 21).

Exercise 1.10 Define a function `removeFst` that removes the first occurrence of an integer *m* from a list of integers. If *m* does not occur in the list, the list remains unchanged.

Example 1.11 We define a function that sorts a list of integers in order of increasing size, by means of the following algorithm:

- an empty list is already sorted.
- if a list is non-empty, we put its minimum in front of the result of sorting the list that results from removing its minimum.

This is implemented as follows:

```
srtInts :: [Int] -> [Int]
srtInts [] = []
srtInts xs = m : (srtInts (removeFst m xs)) where m = mnmInt xs
```

Here `removeFst` is the function you defined in Exercise 1.10. Note that the second clause is invoked when the first one does not apply, i.e., when the argument of `srtInts` is not empty. This ensures that `mnmInt xs` never gives rise to an error.

Note the use of a `where` construction for the local definition of an auxiliary function.

Remark. Haskell has two ways to locally define auxiliary functions, `where` and `let` constructions. The `where` construction is illustrated in Example 1.11. This can also expressed with `let`, as follows:

```
srtInts' :: [Int] -> [Int]
srtInts' [] = []
srtInts' xs = let
                m = mnmInt xs
              in  m : (srtInts' (removeFst m xs))
```

The `let` construction uses the reserved keywords `let` and `in`. ∎

Example 1.12 Here is a function that calculates the average of a list of integers. The average of m and n is given by $\frac{m+n}{2}$, the average of a list of k integers $n_1,\ldots,n_k$ is given by $\frac{n_1+\cdots+n_k}{k}$. In general, averages are fractions, so the result type of `average` should not be `Int` but the Haskell data-type for fractional numbers, which is `Rational`. There are predefined functions `sum` for the sum of a list of integers, and `length` for the length of a list. The Haskell operation for division `/` expects arguments of type `Rational` (or more precisely, of a type in the class `Fractional`, and `Rational` is in that class), so we need a conversion function for converting `Int`s into `Rational`s. This is done by `toRational`. The function `average` can now be written as:

```
average :: [Int] -> Rational
average [] = error "empty list"
average xs = toRational (sum xs) / toRational (length xs)
```

Again, it is instructive to write our own homemade versions of `sum` and `length`. Here they are:

```
sum' :: [Int] -> Int
sum' [] = 0
sum' (x:xs) = x + sum' xs
```

```
length' :: [a] -> Int
length' [] = 0
length' (x:xs) = 1 + length' xs
```

Note that the type declaration for `length'` contains a variable `a`. This variable ranges over all types, so `[a]` is the type of a list of objects of an arbitrary type `a`. We say that `[a]` is a *type scheme* rather than a type. This way, we can use the same function `length'` for computing the length of a list of integers, the length of a list of characters, the length of a list of strings (lists of characters), and so on.

The type `[Char]` is abbreviated as `String`. Examples of characters are `'a'`, `'b'` (note the single quotes) examples of strings are `"Russell"` and `"Cantor"` (note the double quotes). In fact, `"Russell"` can be seen as an abbreviation of the list

`['R','u','s','s','e','l','l']`.

Exercise 1.13 Write a function `count` for counting the number of occurrences of a character in a string. In Haskell, a character is an object of type `Char`, and a string an object of type `String`, so the type declaration should run: `count :: Char -> String -> Int`.

Exercise 1.14 A function for transforming strings into strings is of type `String -> String`. Write a function `blowup` that converts a string

$$a_1 a_2 a_3 \cdots$$

to

$$a_1 a_2 a_2 a_3 a_3 a_3 \cdots .$$

`blowup "bang!"` should yield `"baannnggggg!!!!!"`. (Hint: use `++` for string concatenation.)

Exercise 1.15 Write a function `srtString :: [String] -> [String]` that sorts a list of strings in alphabetical order.

Example 1.16 Suppose we want to check whether a string `str1` is a prefix of a string `str2`. Then the answer to the question `prefix str1 str2` should be either yes (true) or no (false), i.e., the type declaration for `prefix` should run: `prefix :: String -> String -> Bool`.

Prefixes of a string `ys` are defined as follows:

1. [] is a prefix of ys,
2. if xs is a prefix of ys, then x:xs is a prefix of x:ys,
3. nothing else is a prefix of ys.

Here is the code for prefix that implements this definition:

```
prefix :: String -> String -> Bool
prefix [] ys = True
prefix (x:xs) [] = False
prefix (x:xs) (y:ys) = (x==y) && prefix xs ys
```

The definition of prefix uses the Haskell operator && for conjunction.

Exercise 1.17 Write a function substring :: String -> String -> Bool that checks whether str1 is a substring of str2.

The substrings of an arbitrary string ys are given by:

1. if xs is a prefix of ys, xs is a substring of ys,
2. if ys equals y:ys' and xs is a substring of ys', xs is a substring of ys,
3. nothing else is a substring of ys.

1.6 Haskell Types

The basic Haskell types are:

- Int and Integer, to represent integers. Elements of Integer are unbounded. That's why we used this type in the implementation of the prime number test.
- Float and Double represent floating point numbers. The elements of Double have higher precision.
- Bool is the type of Booleans.
- Char is the type of characters.

Note that the name of a type always starts with a capital letter.

To denote arbitrary types, Haskell allows the use of *type variables*. For these, `a`, `b`, ..., are used.

New types can be formed in several ways:

- By list-formation: if `a` is a type, `[a]` is the type of lists over `a`. Examples: `[Int]` is the type of lists of integers; `[Char]` is the type of lists of characters, or strings.
- By pair- or tuple-formation: if `a` and `b` are types, then `(a,b)` is the type of pairs with an object of type `a` as their first component, and an object of type `b` as their second component. Similarly, triples, quadruples, ..., can be formed. If `a`, `b` and `c` are types, then `(a,b,c)` is the type of triples with an object of type `a` as their first component, an object of type `b` as their second component, and an object of type `c` as their third component. And so on (p. 135).
- By function definition: `a -> b` is the type of a function that takes arguments of type `a` and returns values of type `b`.
- By applying a *type constructor*. E.g., `Rational` is the type that results from applying the type constructor `Ratio` to type `Integer`.
- By defining your own data-type from scratch, with a `data` type declaration. More about this in due course.

Pairs will be further discussed in Section 4.5, lists and list operations in Section 4.6.

Operations are procedures for constructing objects of a certain types *b* from ingredients of a type `a`. Now such a procedure can itself be given a type: the type of a transformer from `a` type objects to `b` type objects. The type of such a procedure can be declared in Haskell as `a -> b`.

If a function takes two string arguments and returns a string then this can be viewed as a two-stage process: the function takes a first string and returns a transformer from strings to strings. It then follows that the type is `String -> (String -> String)`, which can be written as `String -> String -> String`, because of the Haskell convention that `->` associates to the right.

Exercise 1.18 Find expressions with the following types:

1. `[String]`
2. `(Bool,String)`

3. `[(Bool,String)]`
4. `([Bool],String)`
5. `Bool -> Bool`

Test your answers by means of the interpreter command `:t`.

Exercise 1.19 Use the interpreter command `:t` to find the types of the following predefined functions:

1. `head`
2. `last`
3. `init`
4. `fst`
5. `(++)`
6. `flip`
7. `flip (++)`

Next, supply these functions with arguments of the expected types, and try to guess what these functions do.

1.7 The Prime Factorization Algorithm

Let n be an arbitrary natural number > 1. A *prime factorization* of n is a list of prime numbers $p_1, \ldots, p_j$ with the property that $p_1 \cdot \cdots \cdot p_j = n$. We will show that a prime factorization of every natural number $n > 1$ exists by producing one by means of the following method of splitting off prime factors:

$$\text{WHILE } n \neq 1 \text{ DO BEGIN } \quad p := \text{LD}(n);\ n := \frac{n}{p} \quad \text{END}$$

Here $:=$ denotes *assignment* or the act of giving a variable a new value. As we have seen, $\text{LD}(n)$ exists for every n with $n > 1$. Moreover, we have seen that $\text{LD}(n)$ is always prime. Finally, it is clear that the procedure terminates, for every round through the loop will decrease the size of n.

So the algorithm consists of splitting off primes until we have written n as $n = p_1 \cdots p_j$, with all factors prime. To get some intuition about how the procedure works, let us see what it does for an example case, say $n = 84$. The original assignment to n is called n_0; successive assignments to n and p are called $n_1, n_2, \ldots$ and $p_1, p_2, \ldots$.

$$\begin{array}{lll} & & n_0 = 84 \\ n_0 \neq 1 & p_1 = 2 & n_1 = 84/2 = 42 \\ n_1 \neq 1 & p_2 = 2 & n_2 = 42/2 = 21 \\ n_2 \neq 1 & p_3 = 3 & n_3 = 21/3 = 7 \\ n_3 \neq 1 & p_4 = 7 & n_4 = 7/7 = 1 \\ n_4 = 1 & & \end{array}$$

This gives $84 = 2^2 \cdot 3 \cdot 7$, which is indeed a prime factorization of 84.

The following code gives an implementation in Haskell, collecting the prime factors that we find in a list. The code uses the predefined Haskell function `div` for integer division.

```
factors :: Integer -> [Integer]
factors n | n < 1     = error "argument not positive"
          | n == 1    = []
          | otherwise = p : factors (div n p) where p = ld n
```

If you load the code for this chapter, you can try this out as follows:

```
GS> factors 84
[2,2,3,7]
GS> factors 55794083012669896096741539O
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71]
```

1.8 The `map` and `filter` Functions

Haskell allows some convenient abbreviations for lists: `[4..20]` denotes the list of integers from 4 through 20, `['a'..'z']` the list of all lower case letters, `"abcdefghijklmnopqrstuvwxyz"`. The call `[5..]` generates an infinite list of integers starting from 5. And so on.

If you use the interpreter command `:t` to find the type of the function `map`, you get the following:

```
Prelude> :t map
map :: (a -> b) -> [a] -> [b]
```

The function map takes a function and a list and returns a list containing the results of applying the function to the individual list members.

If f is a function of type a -> b and xs is a list of type [a], then map f xs will return a list of type [b]. E.g., map (^2) [1..9] will produce the list of squares

```
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

You should verify this by trying it out in the interpreter. The use of (^2) for the operation of squaring demonstrates a new feature of Haskell, the construction of sections.

Conversion from Infix to Prefix, Construction of Sections If op is an infix operator, (op) is the prefix version of the operator. Thus, 2^10 can also be written as (^) 2 10. This is a special case of the use of sections in Haskell.

In general, if op is an infix operator, (op x) is the operation resulting from applying op to its right hand side argument, (x op) is the operation resulting from applying op to its left hand side argument, and (op) is the prefix version of the operator (this is like the abstraction of the operator from both arguments).

Thus (^2) is the squaring operation, (2^) is the operation that computes powers of 2, and (^) is exponentiation. Similarly, (>3) denotes the property of being greater than 3, (3>) the property of being smaller than 3, and (>) is the prefix version of the 'greater than' relation.

The call map (2^) [1..10] will yield

```
[2, 4, 8, 16, 32, 64, 128, 256, 512, 1024]
```

If *p* is a property (an operation of type a -> Bool) and xs is a list of type [a], then map p xs will produce a list of type Bool (a list of truth values), like this:

```
Prelude> map (>3) [1..9]
[False, False, False, True, True, True, True, True, True]
Prelude>
```

The function map is predefined in Haskell, but it is instructive to give our own version:

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Note that if you try to load this code, you will get an error message:

```
Definition of variable "map" clashes with import.
```

The error message indicates that the function name `map` is already part of the name space for functions, and is not available anymore for naming a function of your own making.

Exercise 1.20 Use `map` to write a function `lengths` that takes a list of lists and returns a list of the corresponding list lengths.

Exercise 1.21 Use `map` to write a function `sumLengths` that takes a list of lists and returns the sum of their lengths.

Another useful function is `filter`, for filtering out the elements from a list that satisfy a given property. This is predefined, but here is a home-made version:

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs) | p x       = x : filter p xs
                | otherwise =     filter p xs
```

Here is an example of its use:

```
GS> filter (>3) [1..10]
[4,5,6,7,8,9,10]
```

Example 1.22 Here is a program `primes0` that filters the prime numbers from the infinite list `[2..]` of natural numbers:

```
primes0 :: [Integer]
primes0 = filter prime0 [2..]
```

This produces an infinite list of primes. (Why infinite? See Theorem 3.33.) The list can be interrupted with 'Control-C'.

Example 1.23 Given that we can produce a list of primes, it should be possible now to improve our implementation of the function LD. The function `ldf` used in the definition of `ld` looks for a prime divisor of n by checking $k|n$ for all k with $2 \leqslant k \leqslant \sqrt{n}$. In fact, it is enough to check $p|n$ for the *primes* p with $2 \leqslant p \leqslant \sqrt{n}$. Here are functions `ldp` and `ldpf` that perform this more efficient check:

```
ldp :: Integer -> Integer
ldp n = ldpf primes1 n

ldpf :: [Integer] -> Integer -> Integer
ldpf (p:ps) n | rem n p == 0 = p
              | p^2 > n      = n
              | otherwise    = ldpf ps n
```

`ldp` makes a call to `primes1`, the list of prime numbers. This is a first illustration of a 'lazy list'. The list is called 'lazy' because we compute only the part of the list that we need for further processing. To define `primes1` we need a test for primality, but that test is itself defined in terms of the function LD, which in turn refers to `primes1`. We seem to be running around in a circle. This circle can be made non-vicious by avoiding the primality test for 2. If it is given that 2 is prime, then we can use the primality of 2 in the LD check that 3 is prime, and so on, and we are up and running.

```
primes1 :: [Integer]
primes1 = 2 : filter prime [3..]

prime :: Integer -> Bool
prime n | n < 1     = error "not a positive integer"
        | n == 1    = False
        | otherwise = ldp n == n
```

Replacing the definition of `primes1` by `filter prime [2..]` creates vicious circularity, with stack overflow as a result (try it out). By running the program `primes1` against `primes0` it is easy to check that `primes1` is much faster.

Exercise 1.24 What happens when you modify the defining equation of `ldp` as follows:

```
ldp :: Integer -> Integer
ldp = ldpf primes1
```

Can you explain?

1.9 Haskell Equations and Equational Reasoning

The Haskell equations `f x y = ...` used in the definition of a function `f` are genuine mathematical equations. They state that the left hand side and the right hand side of the equation have the same value. This is *very* different from the use of `=` in imperative languages like C or Java. In a C or Java program, the statement `x = x*y` does *not mean* that x and $x*y$ have the same value, but rather it is a command to throw away the old value of x and put the value of $x*y$ in its place. It is a so-called *destructive assignment statement*: the old value of a variable is destroyed and replaced by a new one.

Reasoning about Haskell definitions is a lot easier than reasoning about programs that use destructive assignment. In Haskell, standard reasoning about mathematical equations applies. E.g., after the Haskell declarations `x = 1` and `y = 2`, the Haskell declaration `x = x + y` will raise an error `"x" multiply defined`. Because `=` in Haskell has the meaning "is by definition equal to", while redefinition is forbidden, reasoning about Haskell functions is standard equational reasoning. Let's try this out on a simple example.

```
a = 3
b = 4
f :: Integer -> Integer -> Integer
f x y = x^2 + y^2
```

To evaluate `f a (f a b)` by equational reasoning, we can proceed as

follows:

$$\begin{aligned} f\ a\ (f\ a\ b) &= f\ a\ (a^2 + b^2) \\ &= f\ 3\ (3^2 + 4^2) \\ &= f\ 3\ (9 + 16) \\ &= f\ 3\ 25 \\ &= 3^2 + 25^2 \\ &= 9 + 625 \\ &= 634 \end{aligned}$$

The rewriting steps use standard mathematical laws and the Haskell definitions of a, b, f. In fact, when running the program we get the same outcome:

```
GS> f a (f a b)
634
GS>
```

Remark. We already encountered definitions where the function that is being defined occurs on the right hand side of an equation in the definition. Here is another example:

```
g :: Integer -> Integer
g 0 = 1
g x = 2 * g (x-1)
```

Not everything that is allowed by the Haskell syntax makes semantic sense, however. The following definitions, although syntactically correct, do not properly define functions:

```
h1 :: Integer -> Integer
h1 0 = 0
h1 x = 2 * (h1 x)

h2 :: Integer -> Integer
h2 0 = 0
h2 x = h2 (x+1)
```

The problem is that for values other than 0 the definition of `h1` does not give a recipe for computing a value, and similary for `h2`, for values greater than 0. This matter will be taken up in Chapter 7. ∎

1.10 Further Reading

The standard Haskell operations are defined in the file *Prelude.hs*, which is described in the Haskell report (`http://www.haskell.org/onlinereport/haskell2010/`).

In case Exercise 1.19 has made you curious, the definitions of these example functions can all be found in *Prelude.hs*. If you want to quickly learn a lot about how to program in Haskell, you should get into the habit of consulting this file regularly. The definitions of all the standard operations are *open source code*, and are there for you to learn from. The Haskell *Prelude* may be a bit difficult to read at first, but you will soon get used to the syntax and acquire a taste for the style.

Various tutorials on Haskell can be found on the Internet: see e.g. [HFP96] (`http://www.haskell.org/tutorial/`). Authoritative references for the language are [Jon03] and `http://www.haskell.org/onlinereport/haskell2010/` for the new 2010 standard of the language.

A textbook on Haskell focusing on multimedia applications is [Hud00]. A very practical Haskell textbook is [OGS09]. Other excellent textbooks on functional programming with Haskell are [Tho99] and, at a more advanced level, [Bir98]. A book on discrete mathematics that also uses Haskell as a tool, and with a nice treatment of automated proof checking, is [HO00].

Chapter 2

Talking about Mathematical Objects

Preview

To talk about mathematical objects with ease it is useful to introduce some symbolic abbreviations. These symbolic conventions are meant to better reveal the **structure** of our mathematical statements. This chapter concentrates on a few (in fact: seven), simple words or phrases that are essential to the mathematical vocabulary: *not*, *if*, *and*, *or*, *if and only if*, *for all* and *for some*. We will introduce symbolic shorthands for these words, and we look in detail at how these building blocks are used to construct the logical patterns of sentences. After having isolated the logical key ingredients of the mathematical vernacular, we can systematically relate definitions in terms of these logical ingredients to implementations, thus building a bridge between logic and computer science.

The use of symbolic abbreviations in specifying algorithms makes it easier to take the step from definitions to the procedures that implement those definitions. In a similar way, the use of symbolic abbreviations in making mathematical statements makes it easier to construct proofs of those statements. Chances are that you are more at ease with programming than with proving things. However that may be, in the chapters to follow you will get the opportunity to improve your skills in both of these activities and to find out more about the way in which they are related.

```
module TAMO

where
-- use hugs -98
-- or use ghci -XFlexibleInstances
```

Caution: the code of this chapter use a slight extension of the Haskell standard syntax (see page 45). Use `hugs -98` or `ghci -XFlexibleInstances` to run it.

2.1 Logical Connectives and their Meanings

Goal To understand how the meanings of statements using connectives can be described by explaining how the truth (or falsity) of the statement depends on the truth (or falsity) of the smallest parts of this statement. This understanding leads directly to an implementation of the logical connectives as truth functional procedures.

In ordinary life, there are many statements that do not have a definite truth value, for example 'Barnett Newman's *Who is Afraid of Red, Yellow and Blue III* is a beautiful work of art,' or 'Daniel Goldreyer's restoration of *Who is Afraid of Red, Yellow and Blue III* meets the highest standards.'

Fortunately the world of mathematics differs from the Amsterdam Stedelijk Museum of Modern Art in the following respect. In the world of mathematics, things are so much clearer that many mathematicians adhere to the following slogan:

> every statement that makes mathematical sense is either true or false.

The idea behind this is that (according to the adherents) the world of mathematics exists independently of the mind of the mathematician. Doing mathematics is the activity of exploring this world. In proving new theorems one discovers new facts about the world of mathematics, in solving exercises one rediscovers known facts for oneself. (Solving problems in a mathematics textbook is like visiting famous places with a tourist guide.)

This belief in an independent world of mathematical fact is called Platonism, after the Greek philosopher Plato, who even claimed that our everyday physical world is somehow an image of this ideal mathematical world. A mathematical Platonist holds that every statement that makes

mathematical sense *has exactly one* of the two truth values. Of course, a Platonist would concede that we may not know which value a statement has, for mathematics has numerous open problems. Still, a Platonist would say that the true answer to an open problem in mathematics like 'Are there infinitely many Mersenne primes?' (Example 3.40 from Chapter 3) is either 'yes' or 'no'. The Platonists would immediately concede that nobody may *know* the true answer, but that, they would say, is an altogether different matter.

Of course, matters are not quite this clear-cut, but the situation is certainly a lot better than in the Amsterdam Stedelijk Museum. In the first place, it may not be immediately obvious which statements make mathematical sense (see Example 4.5). In the second place, you don't have to be a Platonist to do mathematics. Not every working mathematician agrees with the statement that the world of mathematics exists independently of the mind of the mathematical discoverer. The Dutch mathematician Brouwer (1881–1966) and his followers have argued instead that mathematical reality has no independent existence, but is *created* by the working mathematician. According to Brouwer the foundation of mathematics is in the *intuition* of the mathematical intellect. A mathematical Intuitionist will therefore not accept certain proof rules of classical mathematics, such as proof by contradiction (see Section 3.3), as this relies squarely on Platonist assumptions.

Although we have no wish to pick a quarrel with the intuitionists, in this book we will accept proof by contradiction, and we will in general adhere to the practice of classical mathematics and thus to the Platonist creed.

Connectives In mathematical reasoning, it is usual to employ shorthands for *if* (or: *if...then*), *and*, *or*, *not*. These words are called *connectives*. The word *and* is used to form *conjunctions*, its shorthand $\wedge$ is called the *conjunction* symbol. The word *or* is used to form *disjunctions*, its shorthand $\vee$ is called the *disjunction* symbol. The word *not* is used to form *negations*, its shorthand $\neg$ is called the *negation* symbol. The combination *if...then* produces *implications*; its shorthand $\Rightarrow$ is the *implication* symbol. Finally, there is a phrase less common in everyday conversation, but crucial if one is talking mathematics. The combination *...if and only if ...* produces *equivalences*, its shorthand $\Leftrightarrow$ is called the *equivalence* symbol. These logical connectives are summed up in the following table.

	symbol	*name*
and	$\wedge$	conjunction
or	$\vee$	disjunction
not	$\neg$	negation
if—then	$\Rightarrow$	implication
if, and only if	$\Leftrightarrow$	equivalence

Remark. Do not confuse *if... then* ($\Rightarrow$) on one hand with *thus*, *so*, *therefore* on the other. The difference is that the phrase *if... then* is used to construct conditional statements, while *thus* (*therefore*, *so*) is used to combine statements into pieces of mathematical reasoning (or: mathematical proofs). We will never write $\Rightarrow$ when we want to draw a conclusion from a mathematical statement. The rules of inference, the notion of mathematical proof, and the proper use of the word *thus* are the subject of Chapter 3. ∎

Iff. In mathematical English it is usual to abbreviate *if, and only if* to *iff*. We will also use $\Leftrightarrow$ as a symbolic abbreviation. Sometimes the phrase *just in case* is used with the same meaning.

The following describes, for every connective separately, how the truth value of a compound using the connective is determined by the truth values of its components. For most connectives, this is rather obvious. The cases for $\Rightarrow$ and $\vee$ have some peculiar difficulties.

The letters P and Q are used for arbitrary statements. We use **t** for 'true', and **f** for 'false'. The set $\{\mathbf{t},\mathbf{f}\}$ is the set of *truth values*.

Haskell uses `True` and `False` for the truth values. Together, these form the type `Bool`. This type is predefined in Haskell as follows:

```
data Bool    = False | True
```

Negation

An expression of the form $\neg P$ (*not P*, *it is not the case that P*, etc.) is called the *negation* of P. It is true (has truth value **t**) just in case P is false (has truth value **f**).

In an extremely simple table, this looks as follows:

P	$\neg P$
t	**f**
f	**t**

This table is called the *truth table* of the negation symbol.

The implementation of the standard Haskell function `not` reflects this truth table:

```
not          :: Bool -> Bool
not True      = False
not False     = True
```

This definition is part of *Prelude.hs*, the file that contains the predefined Haskell functions.

Conjunction

The expression $P \wedge Q$ ((*both*) P *and* Q) is called the *conjunction* of P and Q. P and Q are called *conjuncts* of $P \wedge Q$. The conjunction $P \wedge Q$ is true iff P and Q are both true.

Truth table of the conjunction symbol:

P	Q	$P \wedge Q$
t	**t**	**t**
t	**f**	**f**
f	**t**	**f**
f	**f**	**f**

This is reflected in definition of the Haskell function for conjunction, `&&` (also from *Prelude.hs*):

```
(&&)  :: Bool -> Bool -> Bool
False && x    = False
True  && x    = x
```

What this says is: if the first argument of a conjunction evaluates to false, then the conjunction evaluates to false; if the first argument evaluates to true, then the conjunction gets the same value as its second argument. The reason that the type declaration has `(&&)` instead of `&&` is that `&&` is an *infix* operator, and `(&&)` is its *prefix* counterpart (see page 21).

Disjunction

The expression $P \vee Q$ (P *or* Q) is called the *disjunction* of P and Q. P and Q are the *disjuncts* of $P \vee Q$.

The interpretation of disjunctions is not always straightforward. English has *two* disjunctions: (i) the *inclusive* version, that counts a disjunction as true also in case both disjuncts are true, and (ii) the *exclusive* version *either... or*, that doesn't.

Remember: The symbol $\vee$ will *always* be used for the **inclusive** version of *or*.

Even with this problem out of the way, difficulties may arise.

Example 2.1 No one will doubt the truth of the following:

for every integer x, $x < 1$ or $0 < x$.

However, acceptance of this brings along acceptance of every instance. E.g., for $x := 1$:[1]

$$1 < 1 \text{ or } 0 < 1.$$

Some people do not find this acceptable or true, or think this to make no sense at all since something better can be asserted, viz., that $0 < 1$. In mathematics with the inclusive version of $\vee$, you'll have to live with such a peculiarity.

The truth table of the disjunction symbol $\vee$ now looks as follows.

P	Q	$P \vee Q$
t	**t**	**t**
t	**f**	**t**
f	**t**	**t**
f	**f**	**f**

Here is the Haskell definition of the disjunction operation. Disjunction is rendered as `||` in Haskell.

```
(||)  :: Bool -> Bool -> Bool
False || x   = x
True  || x   = True
```

[1] := means: 'is by definition equal to'.

What this means is: if the first argument of a disjunction evaluates to false, then the disjunction gets the same value as its second argument. If the first argument of a disjunction evaluates to true, then the disjunction evaluates to true.

Exercise 2.2 Make up the truth table for the *exclusive* version of *or*.

Implication

An expression of the form $P \Rightarrow Q$ (*if P, then Q*; *Q if P*) is called the *implication* of P and Q. P is the *antecedent* of the implication and Q the *consequent*.

The truth table of $\Rightarrow$ is perhaps the only surprising one. However, it can be motivated quite simply using an example like the following. No one will disagree that for every natural number n,

$$5 < n \Rightarrow 3 < n.$$

Therefore, the implication must hold in particular for n equal to 2, 4 and 6. But then, an implication should be *true* if

- both antecedent and consequent are false ($n = 2$),
- antecedent false, consequent true ($n = 4$),

  and
- both antecedent and consequent true ($n = 6$).

Of course, an implication should be false in the only remaining case that the antecedent is true and the consequent false. This accounts for the following truth table.

P	Q	$P \Rightarrow Q$
t	**t**	**t**
t	**f**	**f**
f	**t**	**t**
f	**f**	**t**

If we want to implement implication in Haskell, we can do so in terms of `not` and `||`. It is convenient to introduce an infix operator `==>` for this. The number `1` in the `infix` declaration indicates the binding power (binding power 0 is lowest, 9 is highest). A declaration of an infix operator together with an indication of its binding power is called a *fixity declaration*.

```
infix 1 ==>

(==>) :: Bool -> Bool -> Bool
x ==> y = (not x) || y
```

It is also possible to give a direct definition:

```
(==>) :: Bool -> Bool -> Bool
True  ==> x = x
False ==> x = True
```

Trivially True Implications. Note that implications with antecedent false and those with consequent true are true. For instance, because of this, the following two sentences must be counted as true: *if my name is Napoleon, then the decimal expansion of* π *contains the sequence 7777777*, and: *if the decimal expansion of* π *contains the sequence 7777777, then strawberries are red*.

Implications with one of these two properties (no matter what the values of parameters that may occur) are dubbed *trivially* true. In what follows there are quite a number of facts that are trivial in this sense that may surprise the beginner. One is that the *empty set* $\emptyset$ is included in *every* set (cf. Theorem 4.9 p. 124).

Remark. The word *trivial* is often abused. Mathematicians have a habit of calling things trivial when they are reluctant to prove them. We will try to avoid this use of the word. The justification for calling a statement trivial resides in the psychological fact that a proof of that statement immediately comes to mind. Whether a proof of something comes to your mind will depend on your training and experience, so what is trivial in this sense is (to some extent) a personal matter. When we are reluctant to prove a statement, we will sometimes ask you to prove it as an exercise. ∎

Implication and Causality. The mathematical use of implication does not always correspond to what you are used to. In daily life you will usually require a certain causal dependence between antecedent and consequent of an implication. (This is the reason the previous examples look funny.) In mathematics, such a causality usually will be present, but this is quite

unnecessary for the interpretation of an implication: the truth table tells the complete story. (In this section in particular, causality usually will be absent.) However, in a few cases, natural language use surprisingly corresponds with truth table-meaning. E.g., *I'll be dead if Bill will not show up* must be interpreted (if uttered by someone healthy) as strong belief that Bill will indeed turn up.[2]

Converse and Contrapositive. The *converse* of an implication $P \Rightarrow Q$ is $Q \Rightarrow P$; its *contrapositive* is $\neg Q \Rightarrow \neg P$. The converse of a true implication does not need to be true, but its contrapositive is true iff the implication is. Cf. Theorem 2.10, p. 46.

Necessary and Sufficient Conditions. The statement P is called a *sufficient condition* for Q and Q a *necessary condition* for P if the implication $P \Rightarrow Q$ holds.

An implication $P \Rightarrow Q$ can be expressed in a mathematical text in a number of ways:

1. if P, then Q,
2. Q if P,
3. P only if Q,
4. Q whenever P,
5. P is sufficient for Q,
6. Q is necessary for P.

Equivalence

The expression $P \Leftrightarrow Q$ (P *iff* Q) is called the *equivalence* of P and Q. P and Q are the *members* of the equivalence. The truth table of the equivalence symbol is unproblematic once you realize that an equivalence $P \Leftrightarrow Q$ amounts to the conjunction of two implications $P \Rightarrow Q$ and $Q \Rightarrow P$ taken together. (It is sometimes convenient to write $Q \Rightarrow P$ as $P \Leftarrow Q$.) The outcome is that an equivalence must be true iff its members have the same truth value.

Table:

[2] 'If Bill will not show up, then I am a Dutchman', has the same meaning, when uttered by a native speaker of English. What it means when uttered by one of the authors of this book, we are not sure.

P	Q	$P \Leftrightarrow Q$
t	**t**	**t**
t	**f**	**f**
f	**t**	**f**
f	**f**	**t**

From the discussion under implication it is clear that P is called a condition that is both *necessary* and *sufficient* for Q if $P \Leftrightarrow Q$ is true.

There is no need to add a definition of a function for equivalence to Haskell. The type `Bool` is in class `Eq`, which means that an equality relation is predefined on it. But equivalence of propositions is nothing other than equality of their truth values. Still, it is useful to have a synonym:

```
infix 1 <=>

(<=>) :: Bool -> Bool -> Bool
x <=> y = x == y
```

Example 2.3 When you are asked to prove something of the form P iff Q it is often convenient to separate this into its two parts $P \Rightarrow Q$ and $P \Leftarrow Q$. The 'only if' part of the proof is the proof of $P \Rightarrow Q$ (for $P \Rightarrow Q$ means the same as P only if Q), and the 'if' part of the proof is the proof of $P \Leftarrow Q$ (for $P \Leftarrow Q$ means the same as $Q \Rightarrow P$, which in turn means the same as P, if Q).

Exercise 2.4 Check that the truth table for exclusive *or* from Exercise 2.2 is equivalent to the table for $\neg(P \Leftrightarrow Q)$. Conclude that the Haskell implementation of the function `<+>` for exclusive or in the frame below is correct.

```
infixr 2 <+>

(<+>) :: Bool -> Bool -> Bool
x <+> y = x /= y
```

The logical connectives $\wedge$ and $\vee$ are written in infix notation. Their Haskell counterparts, `&&` and `||` are also infix. Thus, if `p` and `q` are

expressions of type `Bool`, then `p && q` is a correct Haskell expression of type `Bool`. If one wishes to write this in prefix notation, this is also possible, by putting parentheses around the operator: `(&&) p q`.

Although you will probably never find more than 3–5 connectives occurring in one mathematical statement, if you insist you can use as many connectives as you like. Of course, by means of parentheses you should indicate the way your expression was formed.

For instance, look at the formula

$$\neg P \wedge ((P \Rightarrow Q) \Leftrightarrow \neg(Q \wedge \neg P)).$$

Using the truth tables, you can determine its truth value if truth values for the components P and Q have been given. For instance, if P has value **t** and Q has value **f**, then $\neg P$ has **f**, $P \Rightarrow Q$ becomes **f**, $Q \wedge \neg P$: **f**; $\neg(Q \wedge \neg P)$: **t**; $(P \Rightarrow Q) \Leftrightarrow \neg(Q \wedge \neg P)$: **f**, and the displayed expression thus has value **f**. This calculation can be given immediately under the formula, beginning with the values given for P and Q. The final outcome is located under the conjunction symbol $\wedge$, which is the main connective of the expression.

$\neg$	P	$\wedge$	$((P$	$\Rightarrow$	$Q)$	$\Leftrightarrow$	$\neg$	$(Q$	$\wedge$	$\neg$	$P))$
⋮	**t**	⋮	**t**	⋮	**f**	⋮	⋮	**f**	⋮	⋮	**t**
f		⋮		**f**		⋮	⋮		⋮	**f**	
		⋮				⋮	⋮		**f**		
		⋮				⋮	**t**				
		⋮				**f**					
		f									

In compressed form, this looks as follows:

$\neg$	P	$\wedge$	$((P$	$\Rightarrow$	$Q)$	$\Leftrightarrow$	$\neg$	$(Q$	$\wedge$	$\neg$	$P))$
f	**t**	**f**	**t**	**f**	**f**	**f**	**t**	**f**	**f**	**f**	**t**

Alternatively, one might use a computer to perform the calculation.

```
p = True
q = False

formula1 = (not p) && (p ==> q) <=> not (q && (not p))
```

After loading the file with the code of this chapter, you should be able to do:

```
TAMO> formula1
False
```

Note that `p` and `q` are defined as constants, with values `True` and `False`, respectively, so that the occurrences of `p` and `q` in the expression `formula1` are evaluated as these truth values. The rest of the evaluation is then just a matter of applying the definitions of `not`, `&&`, `<=>` and `==>`.

2.2 Logical Validity and Related Notions

Goal To grasp the concepts of logical validity and logical equivalence, to learn how to use truth tables in deciding questions of validity and equivalence, and in the handling of negations, and to learn how the truth table method for testing validity and equivalence can be implemented.

Logical Validities. There are propositional formulas that receive the value **t** no matter what the values of the occurring letters. Such formulas are called (logically) *valid*.

Examples of logical validities are: $P \Rightarrow P$, $P \vee \neg P$, and $P \Rightarrow (Q \Rightarrow P)$.

Truth Table of an Expression. If an expression contains n letters $P,Q,\ldots$, then there are 2^n possible distributions of the truth values between these letters. The 2^n-row table that contains the calculations of these values is the *truth table* of the expression.

If all calculated values are equal to **t**, then your expression, by definition, is a validity.

Example 2.5 (Establishing Logical Validity by Means of a Truth Table)

The following truth table shows that $P \Rightarrow (Q \Rightarrow P)$ is a logical validity.

P	$\Rightarrow$	$(Q$	$\Rightarrow$	$P)$
t	**t**	**t**	**t**	**t**
t	**t**	**f**	**t**	**t**
f	**t**	**t**	**f**	**f**
f	**t**	**f**	**t**	**f**

To see how we can implement the validity check in Haskell, look at the implementation of the evaluation `formula1` again, and add the following definition of `formula2`:

```
formula2 p q = ((not p) && (p ==> q) <=> not (q && (not p)))
```

To see the difference between the two definitions, let us check their types:

```
TAMO> :t formula1
formula1 :: Bool
TAMO> :t formula2
formula2 :: Bool -> Bool -> Bool
TAMO>
```

The difference is that the first definition is a complete proposition (type `Bool`) in itself, while the second still needs two arguments of type `Bool` before it will return a truth value.

In the definition of `formula1`, the occurrences of `p` and `q` are interpreted as *constants*, of which the values are given by previous definitions. In the definition of `formula2`. the occurrences of `p` and `q` are interpreted as *variables* that represent the arguments when the function gets called.

A propositional formula in which the proposition letters are interpreted as variables can in fact be considered as a *propositional function* or *Boolean function* or *truth function*. If just one variable, say p occurs in it, then it is a function of type `Bool -> Bool` (takes a Boolean, returns a Boolean). If two variables occur in it, say p and q, then it is a function of type `Bool -> Bool -> Bool` (takes Boolean, then takes another Boolean, and returns a Boolean). If three variables occur in it, then it is of type `Bool -> Bool -> Bool -> Bool`, and so on.

In the validity check for a propositional formula, we treat the proposition letters as arguments of a propositional function, and we check whether evaluation of the function yields true for every possible combination of the arguments (that is the essence of the truth table method for checking validity).

Here is the case for propositions with one proposition letter (type `Bool -> Bool`).

```
valid1 :: (Bool -> Bool) -> Bool
valid1 bf =  (bf True) && (bf False)
```

The validity check for Boolean functions of type `Bool -> Bool` is suited to test functions of just one variable. An example is the formula $P \vee \neg P$ that expresses the principle of excluded middle (or, if you prefer a Latin name, *tertium non datur*, for: there is no third possibility). Here is its implementation in Haskell:

```
excluded_middle :: Bool -> Bool
excluded_middle p = p || not p
```

To check that this is valid by the truth table method, one should consider the two cases $P := \mathbf{t}$ and $P := \mathbf{f}$, and ascertain that the principle yields $\mathbf{t}$ in both of these cases. This is precisely what the validity check `valid1` does: it yields `True` precisely when applying the boolean function `bf` to `True` yields `True` and applying `bf` to `False` yields `True`. Indeed, we get:

```
TAMO> valid1 excluded_middle
True
```

Here is the validity check for propositional functions with two proposition letters, Such propositional functions have type `Bool -> Bool -> Bool`), and need a truth table with four rows to check their validity, as there are four cases to check.

```
valid2 :: (Bool -> Bool -> Bool)  -> Bool
valid2 bf =   (bf True  True)
           && (bf True  False)
           && (bf False True)
           && (bf False False)
```

Again, it is easy to see that this is an implementation of the truth table method for validity checking. Try this out on $P \Rightarrow (Q \Rightarrow P)$ and on $(P \Rightarrow Q) \Rightarrow P$, and discover that the bracketing matters:

```
form1 p q = p ==> (q ==> p)
form2 p q = (p ==> q) ==> p
```

```
TAM0> valid2 form1
True
TAM0> valid2 form2
False
```

The propositional function `formula2` that was defined above is also of the right argument type for `valid2`:

```
TAM0> valid2 formula2
False
```

It should be clear how the notion of validity is to be implemented for propositional functions with more than two propositional variables. Writing out the full tables becomes a bit irksome, so we are fortunate that Haskell offers an alternative. We demonstrate it in `valid3` and `valid4`.

```
valid3 :: (Bool -> Bool -> Bool -> Bool) -> Bool
valid3 bf = and [ bf p q r | p <- [True,False],
                             q <- [True,False],
                             r <- [True,False]]

valid4 :: (Bool -> Bool -> Bool -> Bool -> Bool) -> Bool
valid4 bf = and [ bf p q r s | p <- [True,False],
                               q <- [True,False],
                               r <- [True,False],
                               s <- [True,False]]
```

The condition `p <- [True,False]`, for "*p* is an element of the list consisting of the two truth values", is an example of *list comprehension* (page 116).

The definitions make use of Haskell list notation, and of the predefined function `and` for generalized conjunction. An example of a list of Booleans in Haskell is `[True,True,False]`. Such a list is said to be of type `[Bool]`. If `list` is a list of Booleans (an object of type `[Bool]`), then `and list` gives `True` in case all members of `list` are true, `False` otherwise. For example, `and [True,True,False]` gives `False`, but `and [True,True,True]` gives `True`. Further details about working with lists can be found in Sections 4.6 and 7.5.

Leaving out Parentheses. We agree that $\wedge$ and $\vee$ bind more strongly than $\Rightarrow$ and $\Leftrightarrow$. Thus, for instance, $P \wedge Q \Rightarrow R$ stands for $(P \wedge Q) \Rightarrow R$ (and *not* for $P \wedge (Q \Rightarrow R)$).

Operator Precedence in Haskell In Haskell, the convention is not quite the same, for `||` has operator precedence 2, `&&` has operator precedence 3, and `==` has operator precedence 4, which means that `==` binds more strongly than `&&`, which in turn binds more strongly than `||`. The operators that we added, `==>` and `<=>`, follow the logic convention: they bind less strongly than `&&` and `||`.

Logically Equivalent. Two formulas are called (logically) *equivalent* if, no matter the truth values of the letters $P, Q, \ldots$ occurring in these formulas, the truth values obtained for them are the same. This can be checked by constructing a truth table (see Example 2.6).

Example 2.6 (The First Law of De Morgan)

$\neg$	$(P$	$\wedge$	$Q)$		$(\neg$	P	$\vee$	$\neg$	$Q)$
f	**t**	**t**	**t**		**f**	**t**	**f**	**f**	**t**
t	**t**	**f**	**f**		**f**	**t**	**t**	**t**	**f**
t	**f**	**f**	**t**		**t**	**f**	**t**	**f**	**t**
t	**f**	**f**	**f**		**t**	**f**	**t**	**t**	**f**

The outcome of the calculation shows that the formulas are equivalent: note that the column under the $\neg$ of $\neg(P \wedge Q)$ coincides with that under the $\vee$ of $\neg P \vee \neg Q$.

Notation: $\Phi \equiv \Psi$ indicates that Φ and Ψ are equivalent.[3] Using this notation, we can say that the truth table of Example 2.6 shows that $\neg(P \wedge Q) \equiv (\neg P \vee \neg Q)$.

Example 2.7 (De Morgan Again)
The following truth table shows that $\neg(P \wedge Q) \Leftrightarrow (\neg P \vee \neg Q)$ is a logical validity, which establishes that $\neg(P \wedge Q) \equiv (\neg P \vee \neg Q)$.

$\neg$	$(P$	$\wedge$	$Q)$	$\Leftrightarrow$	$(\neg$	P	$\vee$	$\neg$	$Q)$
f	**t**	**t**	**t**	**t**	**f**	**t**	**f**	**f**	**t**
t	**t**	**f**	**f**	**t**	**f**	**t**	**t**	**t**	**f**
t	**f**	**f**	**t**	**t**	**t**	**f**	**t**	**f**	**t**
t	**f**	**f**	**f**	**t**	**t**	**f**	**t**	**t**	**f**

[3] The Greek alphabet is on p. 419.

Example 2.8 A pixel on a computer screen is a dot on the screen that can be either *on* (i.e., visible) or *off* (i.e., invisible). We can use 1 for *on* and 0 for *off*. Turning pixels in a given area on the screen off or on creates a screen pattern for that area. The screen pattern of an area is given by a list of bits (0s or 1s). Such a list of bits can be viewed as a list of truth values (by equating 1 with **t** and 0 with **f**), and given two bit lists of the same length we can perform bitwise logical operations on them: the bitwise exclusive or of two bit lists of the same length n, say $L = [P_1, \ldots, P_n]$ and $K = [Q_1, \ldots, Q_n]$, is the list $[P_1 \oplus Q_1, \ldots, P_n \oplus Q_n]$, where $\oplus$ denotes exclusive or.

In the implementation of cursor movement algorithms, the cursor is made visible on the screen by taking a bitwise exclusive or between the screen pattern S at the cursor position and the cursor pattern C. When the cursor moves elsewhere, the original screen pattern is restored by taking a bitwise exclusive or with the cursor pattern C again. Exercise 2.9 shows that this indeed restores the original pattern S.

Exercise 2.9 Let $\oplus$ stand for exclusive or. Show, using the truth table from Exercise 2.2, that $(P \oplus Q) \oplus Q$ is equivalent to P.

In Haskell, logical equivalence can be tested as follows. First we give a procedure for propositional functions with 1 parameter:

```
logEquiv1 ::  (Bool -> Bool) -> (Bool -> Bool) -> Bool
logEquiv1 bf1 bf2 =
    (bf1 True  <=> bf2 True) && (bf1 False <=> bf2 False)
```

What this does, for formulas Φ, Ψ with a single propositional variable, is testing the formula $\Phi \Leftrightarrow \Psi$ by the truth table method.

We can extend this to propositional functions with 2, 3 or more parameters, using generalized conjunction. Here are the implementations of `logEquiv2` and `logEquiv3`; it should be obvious how to extend this for truth functions with still more arguments.

```
logEquiv2 :: (Bool -> Bool -> Bool) ->
                    (Bool -> Bool -> Bool) -> Bool
logEquiv2 bf1 bf2 =
  and [(bf1 p q) <=> (bf2 p q)  |  p <- [True,False],
                                   q <- [True,False]]

logEquiv3 :: (Bool -> Bool -> Bool -> Bool) ->
                  (Bool -> Bool -> Bool -> Bool) -> Bool
logEquiv3 bf1 bf2 =
  and [(bf1 p q r) <=> (bf2 p q r) |  p <- [True,False],
                                      q <- [True,False],
                                      r <- [True,False]]
```

Let us redo Exercise 2.9 by computer.

```
formula3 p q = p
formula4 p q = (p <+> q) <+> q
```

Note that the `q` in the definition of `formula3` is needed to ensure that it is a function with two arguments.

```
TAMO> logEquiv2 formula3 formula4
True
```

We can also test this by means of a validity check on $P \Leftrightarrow ((P \oplus Q) \oplus Q)$, as follows:

```
formula5 p q = p <=> ((p <+> q) <+> q)
```

```
TAMO> valid2 formula5
True
```

Warning. Do not confuse $\equiv$ and $\Leftrightarrow$. If Φ and Ψ are formulas, then $\Phi \equiv \Psi$ expresses the statement that Φ and Ψ are equivalent. On the other hand, $\Phi \Leftrightarrow \Psi$ is just another formula. The relation between the two is that the formula $\Phi \Leftrightarrow \Psi$ is logically valid iff it holds that $\Phi \equiv \Psi$. (See Exercise 2.19.) Compare the difference, in Haskell, between `logEquiv2 formula3 formula4` (a true statement about the relation between two formulas), and `formula5` (just another formula).

Instead of giving separate treatments for propositions with one proposition letter (type `Bool -> Bool`), propositions with two proposition letter (type `Bool -> Bool -> Bool`), and so on, it makes sense to give a general treatment. This can be done with Haskell *type classes*. Here is a declaration of a type class for truth functions:

```
class TF p where
  valid :: p -> Bool
  lequiv :: p -> p -> Bool
```

What this says is that if type p is in the class of truth functions, then the property of validity `valid` and the relation of logical equivalence `lequiv` should be defined for type p.

The simplest case of a truth function is the case of a truth function without any propositional variables at all. These are just the constants `True` and `False`. Of these, the constant `True` is valid, and logical equivalence between truth constants boils down to equality. Here is the instantiation of that class. The instantiation will have the effect that `True` is classified as valid, but `False` is not, and that two truth values are classified as logically equivalent just in case they are the same.

```
instance TF Bool
 where
  valid  = id
  lequiv f g = f == g
```

The other truth functional types can be defined by recursion. If we know that `p` is the type of a truth function, then surely `Bool -> p` is the type of a truth function. The following declaration uses a slight extension of the Haskell standard syntax (use `hugs -98` or `ghci -XFlexibleInstances` to run the code):

```
instance TF p => TF (Bool -> p)
 where
  valid f = valid (f True) && valid (f False)
  lequiv f g = (f True) `lequiv` (g True)
               && (f False) `lequiv` (g False)
```

Since `p` is a truth functional type, we know we have functions `valid` and `lequiv` available for objects of type `p`.

Note that if `f` has type `Bool -> p`, then `(f True)` and `(f False)` have type `p`. So the extension of `valid` to the type `Bool -> p` is simply this: check whether both `(f True)` and `(f False)` are valid.

Similarly for `lequiv`: just check whether `f` and `g` are equivalent for the two arguments `True` and `False` we can give them.

The following theorem collects a number of useful equivalences. (Of course, P, Q and R can be arbitrary formulas themselves.)

Theorem 2.10 *1.* $P \equiv \neg\neg P$ *(law of double negation),*

2. $P \wedge P \equiv P$; $P \vee P \equiv P$ *(laws of idempotence),*

3. $(P \Rightarrow Q) \equiv \neg P \vee Q$;
$\neg(P \Rightarrow Q) \equiv P \wedge \neg Q$,

4. $(\neg P \Rightarrow \neg Q) \equiv (Q \Rightarrow P)$;
$(P \Rightarrow \neg Q) \equiv (Q \Rightarrow \neg P)$;
$(\neg P \Rightarrow Q) \equiv (\neg Q \Rightarrow P)$ *(laws of contraposition),*

5. $(P \Leftrightarrow Q) \equiv ((P \Rightarrow Q) \wedge (Q \Rightarrow P))$
$\equiv ((P \wedge Q) \vee (\neg P \wedge \neg Q))$,

6. $P \wedge Q \equiv Q \wedge P$; $P \vee Q \equiv Q \vee P$ *(laws of commutativity),*

7. $\neg(P \wedge Q) \equiv \neg P \vee \neg Q$;
$\neg(P \vee Q) \equiv \neg P \wedge \neg Q$ *(DeMorgan laws).*

8. $P \wedge (Q \wedge R) \equiv (P \wedge Q) \wedge R$;
$P \vee (Q \vee R) \equiv (P \vee Q) \vee R$ *(laws of associativity),*

9. $P \wedge (Q \vee R) \equiv (P \wedge Q) \vee (P \wedge R)$;
$P \vee (Q \wedge R) \equiv (P \vee Q) \wedge (P \vee R)$ *(distribution laws),*

Equivalence 8 justifies leaving out parentheses in conjunctions and disjunctions of three or more conjuncts resp., disjuncts. Non-trivial equivalences that often are used in practice are 2, 3 and 9. Note how you can use these to re-write negations: a negation of an implication can be rewritten as a conjunction, a negation of a conjunction (disjunction) is a disjunction (conjunction).

Exercise 2.11 The First Law of De Morgan was proved in Example 2.6. This method was implemented above. Use the method by hand to prove the other parts of Theorem 2.10.

```
test1  = lequiv id (\ p -> not (not p))
test2a = lequiv id (\ p -> p && p)
test2b = lequiv id (\ p -> p || p)
test3a = lequiv (\ p q -> p ==> q) (\ p q -> not p || q)
test3b = lequiv (\ p q -> not (p ==> q)) (\ p q -> p && not q)
test4a = lequiv (\ p q -> not p ==> not q) (\ p q -> q ==> p)
test4b = lequiv (\ p q -> p ==> not q) (\ p q -> q ==> not p)
test4c = lequiv (\ p q -> not p ==> q) (\ p q -> not q ==> p)
test5a = lequiv (\ p q -> p <=> q)
                (\ p q -> (p ==> q) && (q ==> p))
test5b = lequiv (\ p q -> p <=> q)
                (\ p q -> (p && q) || (not p && not q))
test6a = lequiv (\ p q -> p && q) (\ p q -> q && p)
test6b = lequiv (\ p q -> p || q) (\ p q -> q || p)
test7a = lequiv (\ p q -> not (p && q))
                (\ p q -> not p || not q)
test7b = lequiv (\ p q -> not (p || q))
                (\ p q -> not p && not q)
test8a = lequiv (\ p q r -> p && (q && r))
                (\ p q r -> (p && q) && r)
test8b = lequiv (\ p q r -> p || (q || r))
                (\ p q r -> (p || q) || r)
test9a = lequiv (\ p q r -> p && (q || r))
                (\ p q r -> (p && q) || (p && r))
test9b = lequiv (\ p q r ->  p || (q && r))
                (\ p q r -> (p || q) && (p || r))
```

Figure 2.1: Defining the Tests for Theorem 2.10.

We will now demonstrate how one can use the implementation of the logical equivalence tests as a check for Theorem 2.10. Here is a question for you to ponder: does checking the formulas by means of the implemented functions for logical equivalence count as a *proof* of the principles involved? Whatever the answer to this one may be, Figure 2.1 defines the tests for the statements made in Theorem 2.10, by means of *lambda abstraction*

The expression `\ p -> not (not p)` is the Haskell way of referring to the lambda term $\lambda p.\neg\neg p$, the term that denotes the operation of performing a double negation. See Section 2.4.

If you run these tests, you get result `True` for all of them. E.g.:

```
TAM0> test5a
True
```

The next theorem lists some useful principles for reasoning with $\top$ (the proposition that is always true; the Haskell counterpart is `True`) and $\bot$ (the proposition that is always false; the Haskell counterpart of this is `False`).

Theorem 2.12 *1.* $\neg\top \equiv \bot$; $\neg\bot \equiv \top$,

2. $P \Rightarrow \bot \equiv \neg P$,

3. $P \vee \top \equiv \top$; $P \wedge \bot \equiv \bot$ *(dominance laws),*

4. $P \vee \bot \equiv P$; $P \wedge \top \equiv P$ *(identity laws),*

5. $P \vee \neg P \equiv \top$ *(law of excluded middle),*

6. $P \wedge \neg P \equiv \bot$ *(contradiction).*

Exercise 2.13 Implement checks for the principles from Theorem 2.12.

Without proof, we state the following **Substitution Principle**: If Φ and Ψ are equivalent, and Φ' and Ψ' are the results of substituting Ξ for every occurrence of P in Φ and in Ψ, respectively, then Φ' and Ψ' are equivalent. Example 2.14 makes clear what this means.

Example 2.14 From $\neg(P \Rightarrow Q) \equiv P \wedge \neg Q$ plus the substitution principle it follows that

$$\neg(\neg P \Rightarrow Q) \equiv \neg P \wedge \neg Q$$

(by substituting $\neg P$ for P), but also that

$$\neg(a = 2^b - 1 \Rightarrow a \text{ is prime}) \equiv a = 2^b - 1 \wedge a \text{ is not prime}$$

(by substituting $a = 2^b - 1$ for P and *a is prime* for Q).

Exercise 2.15 A propositional contradiction is a formula that yields false for every combination of truth values for its proposition letters. Write Haskell definitions of contradiction tests for propositional functions with one, two and three variables.

Exercise 2.16 Produce useful denials for every sentence of Exercise 2.31. (A denial of Φ is an equivalent of $\neg\Phi$.)

Exercise 2.17 Produce a denial for the statement that $x < y < z$ (where $x, y, z \in \mathbb{R}$).

Exercise 2.18 Show:

1. $(\Phi \Leftrightarrow \Psi) \equiv (\neg\Phi \Leftrightarrow \neg\Psi)$,
2. $(\neg\Phi \Leftrightarrow \Psi) \equiv (\Phi \Leftrightarrow \neg\Psi)$.

Exercise 2.19 Show that $\Phi \equiv \Psi$ is true iff $\Phi \Leftrightarrow \Psi$ is logically valid.

Exercise 2.20 Determine (either using truth tables or Theorem 2.10) which of the following are equivalent, next check your answer by computer:

1. $\neg P \Rightarrow Q$ and $P \Rightarrow \neg Q$,
2. $\neg P \Rightarrow Q$ and $Q \Rightarrow \neg P$,
3. $\neg P \Rightarrow Q$ and $\neg Q \Rightarrow P$,
4. $P \Rightarrow (Q \Rightarrow R)$ and $Q \Rightarrow (P \Rightarrow R)$,
5. $P \Rightarrow (Q \Rightarrow R)$ and $(P \Rightarrow Q) \Rightarrow R$,
6. $(P \Rightarrow Q) \Rightarrow P$ and P,
7. $P \vee Q \Rightarrow R$ and $(P \Rightarrow R) \wedge (Q \Rightarrow R)$.

Exercise 2.21 Answer as many of the following questions as you can:

1. Construct a formula Φ involving the letters P and Q that has the following truth table.

P	Q	Φ
t	**t**	**t**
t	**f**	**t**
f	**t**	**f**
f	**f**	**t**

2. How many truth tables are there for 2-letter formulas altogether?
3. Can you find formulas for all of them?
4. Is there a general method for finding these formulas?
5. What about 3-letter formulas and more?

2.3 Making Symbolic Form Explicit

In a sense, propositional reasoning is not immediately relevant for mathematics. Few mathematicians will ever feel the urge to write down a disjunction of two statements like $3 < 1 \vee 1 < 3$. In cases like this it is clearly "better" to only write down the right-most disjunct.

Fortunately, once variables enter the scene, propositional reasoning suddenly becomes a very useful tool: the connectives turn out to be quite useful for combining *open* formulas. An open formula is a formula with one or more unbound variables in it. Variable binding will be explained below, but here is a first example of a formula with an unbound variable x. A disjunction like $3 < x \vee x < 3$ is (in some cases) a useful way of expressing that $x \neq 3$.

Example. Consider the following (true) sentence:

Between every two rational numbers there is a third one. (2.1)

The property expressed in (2.1) is usually referred to as *density* of the rationals. We will take a systematic look at proving such statements in Chapter 3.

Exercise 2.22 Can you think of an argument showing that statement (2.1) is true?

A Pattern. There is a *logical pattern* underlying sentence (2.1). To make it visible, look at the following, more explicit, formulation. It uses *variables* x,y and z for arbitrary rationals, and refers to the *ordering* $<$ of the set $\mathbb{Q}$ of rational numbers.

For all rational numbers x and z, if $x < z$, then some rational number y exists such that $x < y$ and $y < z$. (2.2)

You will often find '$x < y$ *and* $y < z$' shortened to: $x < y < z$.

Quantifiers Note the words *all* (or: *for all*), *some* (or: *for some*, *some…exists*, *there exists…such that*, etc.). They are called *quantifiers*, and we use the symbols $\forall$ and $\exists$ as shorthands for them.

With these shorthands, plus the shorthands for the connectives that we saw above, and the shorthand $\ldots \in \mathbb{Q}$ for the property of being a rational, we arrive at the following compact symbolic formulation:

$$\forall x{\in}\mathbb{Q}\,\forall z{\in}\mathbb{Q}\ (x < z \ \Rightarrow\ \exists y{\in}\mathbb{Q}\,(x < y \ \wedge\ y < z)\,). \tag{2.3}$$

We will use example (2.3) to make a few points about the proper use of the vocabulary of logical symbols. An expression like (2.3) is called a sentence or a formula. Note that the example formula (2.3) is composite: we can think of it as constructed out of simpler parts. We can picture its structure as in Figure 2.2.

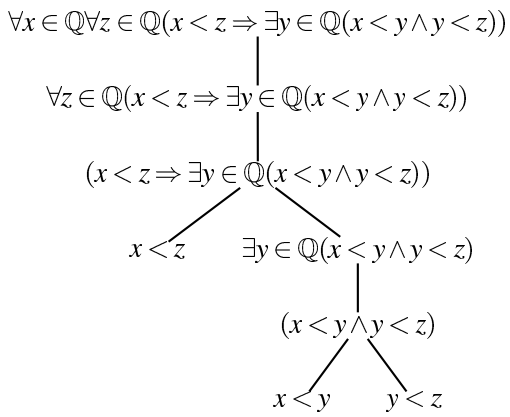

Figure 2.2: Composition of Example Formula from its Sub-formulas.

As the figure shows, the example formula is formed by putting the quantifier prefix $\forall x \in \mathbb{Q}$ in front of the result of putting quantifier prefix $\forall z \in \mathbb{Q}$ in front of a simpler formula, and so on.

The two consecutive universal quantifier prefixes can also be combined into $\forall x,z{\in}\mathbb{Q}$. This gives the phrasing

$$\forall x,z \in \mathbb{Q}(\,x < z \ \Rightarrow\ \exists y{\in}\mathbb{Q}\,(x < y \ \wedge\ y < z)\,).$$

Putting an $\wedge$ between the two quantifiers is incorrect, however. In other words, the expression $\forall x{\in}\mathbb{Q} \wedge \forall z{\in}\mathbb{Q}(x < z \Rightarrow \exists y{\in}\mathbb{Q}(x < y \wedge y < z))$ is considered *un*grammatical. The reason is that the formula part $\forall x \in \mathbb{Q}$ is itself not a formula, but a prefix that turns a formula into a more complex formula. The connective $\wedge$ can only be used to construct a new formula out of two simpler formulas, so $\wedge$ cannot serve to construct a formula from $\forall x \in \mathbb{Q}$ and another formula.

The symbolic version of the density statement uses parentheses. Their function is to indicate the way the expression has been formed and thereby to show the *scope* of operators. The scope of a quantifier-expression is the formula that it combines with to form a more complex formula. The scopes of quantifier-expressions and connectives in a formula are illustrated in the structure tree of that formula. Figure 2.2 shows that the scope of the quantifier-expression $\forall x \in \mathbb{Q}$ is the formula

$$\forall z \in \mathbb{Q}(x < z \Rightarrow \exists y \in \mathbb{Q}\ \ (x < y \wedge y < z)),$$

the scope of $\forall z \in \mathbb{Q}$ is the formula

$$(x < z \Rightarrow \exists y \in \mathbb{Q}(x < y \wedge y < z)),$$

and the scope of $\exists y \in \mathbb{Q}$ is the formula $(x < y \wedge y < z)$.

Exercise 2.23 Give structure trees of the following formulas (we use shorthand notation, and write $A(x)$ as Ax for readability).

1. $\forall x(Ax \Rightarrow (Bx \Rightarrow Cx))$.
2. $\exists x(Ax \wedge Bx)$.
3. $\exists xAx \wedge \exists xBx$.

The expression *for all* (and similar ones) and its shorthand, the symbol $\forall$, is called the *universal quantifier*; the expression *there exists* (and similar ones) and its shorthand, the symbol $\exists$, is called the *existential quantifier*. The letters x, y and z that have been used in combination with them are *variables*. Note that 'for some' is equivalent to 'for at least one'.

Unrestricted and Restricted Quantifiers, Domain of Quantification Quantifiers can occur *unrestricted*: $\forall x(x \geqslant 0)$, $\exists y \forall x(y > x)$, and *restricted*: $\forall x \in A(x \geqslant 0)$, $\exists y \in B(y < a)$ (where A and B are sets).

In the unrestricted case, there should be some *domain of quantification* that often is implicit in the context. E.g., if the context is real analysis, $\forall x$ may mean *for all reals* $x\ldots$, and $\forall f$ may mean *for all real-valued functions* $f\ldots$.

Example 2.24 $\mathbb{R}$ is the set of real numbers. The fact that the $\mathbb{R}$ has no greatest element can be expressed with restricted quantifiers as:

$$\forall x \in \mathbb{R} \exists y \in \mathbb{R}(x < y).$$

If we specify that all quantifiers range over the reals (i.e., if we say that $\mathbb{R}$ is the domain of quantification) then we can drop the explicit restrictions, and we get by with $\forall x \exists y(x < y)$.

The use of restricted quantifiers allows for greater flexibility, for it permits one to indicate different domains for different quantifiers.

Example 2.25

$$\forall x \in \mathbb{R} \forall y \in \mathbb{R}(x < y \Rightarrow \exists z \in \mathbb{Q}\ \ (x < z < y)).$$

Instead of $\exists x(Ax \wedge \ldots)$ one can write $\exists x \in A(\ldots)$. The advantage when all quantifiers are thus restricted is that it becomes immediately clear that the domain is subdivided into different sub domains or types. This can make the logical translation much easier to comprehend.

Remark. We will use standard names for the following domains: $\mathbb{N}$ for the natural numbers, $\mathbb{Z}$ for the integer numbers, $\mathbb{Q}$ for the rational numbers, and $\mathbb{R}$ for the real numbers. More information about these domains can be found in Chapter 8. ∎

Exercise 2.26 Write as formulas with restricted quantifiers:

1. $\exists x \exists y(x \in \mathbb{Q} \wedge y \in \mathbb{Q} \wedge x < y)$.
2. $\forall x(x \in \mathbb{R} \Rightarrow \exists y(y \in \mathbb{R} \wedge x < y))$.
3. $\forall x(x \in \mathbb{Z} \Rightarrow \exists m,n(m \in \mathbb{N} \wedge n \in \mathbb{N} \wedge x = m - n))$.

Exercise 2.27 Write as formulas without restricted quantifiers:

1. $\forall x \in \mathbb{Q} \exists m,n \in \mathbb{Z}(n \neq 0 \wedge x = m/n)$.
2. $\forall x \in F \forall y \in D(Oxy \Rightarrow Bxy)$.

Bound Variables. Quantifier expressions $\forall x$, $\exists y$,... (and their restricted companions) are said to *bind* every occurrence of x, y,... in their scope. If a variable occurs *bound* in a certain expression then the meaning of that expression does not change when all bound occurrences of that variable are replaced by another one.

Example 2.28 $\exists y \in \mathbb{Q}(x < y)$ has the same meaning as $\exists z \in \mathbb{Q}(x < z)$. This indicates that y is bound in $\exists y \in \mathbb{Q}(x < y)$. But $\exists y \in \mathbb{Q}(x < y)$ and $\exists y \in \mathbb{Q}(z < y)$ have different meanings, for the first asserts that there exists a rational number greater than some given number x, and the second that there exists a rational number greater than some given z.

Universal and existential quantifiers are not the only variable binding operators used by mathematicians. There are several other constructs that you are probably familiar with which can bind variables.

Example 2.29 (Summation, Integration.) The expression $\sum_{i=1}^{5} i$ is nothing but a way to describe the number 15 ($15 = 1+2+3+4+5$), and clearly, 15 does in no way depend on i. Use of a different variable does not change the meaning: $\sum_{k=1}^{5} k = 15$. Here are the Haskell versions:

```
Prelude> sum [ i | i <- [1..5] ]
15
Prelude> sum [ k | k <- [1..5] ]
15
```

Similarly, the expression $\int_0^1 x dx$ denotes the number $\frac{1}{2}$ and does not depend on x.

Example 2.30 (Abstraction.) Another way to bind a variable occurs in the *abstraction notation* $\{x \in A \mid P\}$, cf. (4.1), p. 116. The Haskell counterpart to this is list comprehension:

```
[ x | x <- list, property x ]
```

The choice of variable does not matter. The same list is specified by:

```
[ y | y <- list, property y ]
```

The way set comprehension is used to define sets is similar to the way list comprehension is used to define lists, and this is similar again to the way lambda abstraction is used to define functions. See Section 2.4.

Bad Habits. It is not unusual to encounter our example-statement (2.1) displayed as follows.

> *For all rationals x and y, if $x < y$, then both $x < z$ and $z < y$ hold for some rational z.*

Note that the meaning of this is not completely clear. With this expression the true statement that $\forall x,y\in\mathbb{Q}\,\exists z\in\mathbb{Q}\,(x<y \;\Rightarrow\; (x<z\wedge z<y))$ could be meant, but what also could be meant is the false statement that $\exists z\in\mathbb{Q}\,\forall x,y\in\mathbb{Q}\,(x<y \;\Rightarrow\; (x<z\wedge z<y))$.

Putting quantifiers both at the front and at the back of a formula results in ambiguity, for it becomes difficult to determine their scopes. In the worst case the result is an ambiguity between statements that mean entirely different things.

It does not look too well to let a quantifier bind an expression that is not a variable, such as in:

for all numbers n^2+1, ...

Although this habit does not always lead to unclarity, it is better to avoid it, as the result is often rather hard to comprehend. If you insist on quantifying over complex terms, then the following phrasing is suggested: for all numbers *of the form* $n^2+1,\ldots$

Of course, in the implementation language, terms like `n + 1` are important for *pattern matching*.

Translation Problems. It is easy to find examples of English sentences that are hard to translate into the logical vernacular. E.g., in *between two rationals is a third one* it is difficult to discover a universal quantifier and an implication.

Also, note that indefinites in natural language may be used to express universal statements. Consider the sentence *a well-behaved child is a quiet child*. The indefinite articles here may suggest *existential* quantifiers; however, the reading that is clearly meant has the form

$$\forall x\in \mathrm{C}\,(\text{Well-behaved}(x) \;\Rightarrow\; \text{Quiet}(x)).$$

A famous example from philosophy of language is: *if a farmer owns a donkey, he beats it*. Again, in spite of the indefinite articles, the meaning is universal:

$$\forall x\forall y((\text{Farmer}(x)\wedge\text{Donkey}(y)\wedge\text{Own}(x,y))\Rightarrow\text{Beat}(x,y)).$$

In cases like this, translation into a formula reveals the logical meaning that remained hidden in the original phrasing.

In mathematical texts it also occurs quite often that the indefinite article *a* is used to make universal statements. Compare Example 2.43 below, where the following universal statement is made: *A real function is continuous if it satisfies the* ε-δ-*definition.*

Exercise 2.31 Translate into formulas, taking care to express the intended meaning:

1. The equation $x^2+1=0$ has a solution.
2. A largest natural number does not exist.
3. The number 13 is prime (use $d|n$ for 'd divides n').
4. The number n is prime.
5. There are infinitely many primes.

Exercise 2.32 Translate into formulas:

1. Everyone loved Diana. (Use the expression $L(x,y)$ for: x loved y, and the name d for Diana.)
2. Diana loved everyone.
3. Man is mortal. (Use $M(x)$ for 'x is a man', and $M'(x)$ for 'x is mortal'.)
4. Some birds do not fly. (Use $B(x)$ for 'x is a bird' and $F(x)$ for 'x can fly'.)

Exercise 2.33 Translate into formulas, using appropriate expressions for the predicates:

1. Dogs that bark do not bite.
2. All that glitters is not gold.
3. Friends of Diana's friends are her friends.
4. *The limit of $\frac{1}{n}$ as n approaches infinity is zero.

Expressing Uniqueness. If we combine quantifiers with the relation $=$ of identity, we can make definite statements like 'there is precisely one real number x with the property that for any real number y, $xy=y$'. The logical rendering is (assuming that the domain of discussion is $\mathbb{R}$):

$$\exists x(\forall y(x\cdot y=y)\wedge\forall z(\forall y(z\cdot y=y)\Rightarrow z=x)).$$

The first part of this formula expresses that at least one x satisfies the property $\forall y(x\cdot y=y)$, and the second part states that any z satisfying the same property is identical to that x.

The logical structure becomes more transparent if we write P for the property. This gives the following translation for 'precisely one object has property P':

$$\exists x(Px \wedge \forall z(Pz \Rightarrow z = x)).$$

Exercise 2.34 Use the identity symbol $=$ to translate the following sentences:

1. Everyone loved Diana except Charles.
2. Every man adores at least two women.
3. No man is married to more than one woman.

Long ago the philosopher Bertrand Russell has proposed this logical format for the translation of the English definite article. According to his theory of description, the translation of *The King is raging* becomes:

$$\exists x(\text{King}(x) \wedge \forall y(\text{King}(y) \Rightarrow y = x) \wedge \text{Raging}(x)).$$

Exercise 2.35 Use Russell's recipe to translate the following sentences:

1. The King is not raging.
2. The King is loved by all his subjects. (use $K(x)$ for 'x is a King', and $S(x,y)$ for 'x is a subject of y').

Exercise 2.36 Translate the following logical statements back into English.

1. $\exists x \in \mathbb{R}(x^2 = 5)$.
2. $\forall n \in \mathbb{N} \exists m \in \mathbb{N}(n < m)$.
3. $\forall n \in \mathbb{N} \neg \exists d \in \mathbb{N}(1 < d < (2^n + 1) \wedge d | (2^n + 1))$.
4. $\forall n \in \mathbb{N} \exists m \in \mathbb{N}(n < m \wedge \forall p \in \mathbb{N}(p \leqslant n \vee m \leqslant p))$.
5. $\forall \varepsilon \in \mathbb{R}^+ \exists n \in \mathbb{N} \forall m \in \mathbb{N}(m \geqslant n \Rightarrow (|a - a_m| \leqslant \varepsilon))$. ($\mathbb{R}^+$ is the set of positive reals; $a, a_0, a_1, \ldots$ refer to real numbers .)

Remark. Note that translating back and forth between formulas and plain English involves making decisions about a domain of quantification and about the predicates to use. This is often a matter of taste. For instance, how does one choose between $P(n)$ for 'n is prime' and the spelled out

$$n > 1 \wedge \neg \exists d \in \mathbb{N}(1 < d < n \wedge d | n),$$

which expands the definition of being prime? Expanding the definitions of mathematical concepts is not always a good idea. The purpose of introducing

the word *prime* was precisely to hide the details of the definition, so that they do not burden the mind. The art of finding the right mathematical phrasing is to introduce precisely the amount and the kind of complexity that are needed to handle a given problem. ∎

Before we will start looking at the language of mathematics and its conventions in a more systematic way, we will make the link between mathematical definitions and implementations of those definitions.

2.4 Lambda Abstraction

The following description defines a specific function that does not depend at all on x:

> The function that sends x to x^2.

Often used notations are $x \mapsto x^2$ and $\lambda x.x^2$. The expression $\lambda x.x^2$ is called a *lambda term*.

If t is an expression of type b and x is a variable of type a then $\lambda x.t$ is an expression of type $a \to b$, i.e., $\lambda x.t$ denotes a function. This way of defining functions is called lambda abstraction.

Note that *the function that sends y to y^2* (notation $y \mapsto y^2$, or $\lambda y.y^2$) describes the same function as $\lambda x.x^2$.

In Haskell, function definition by lambda abstraction is available. Compare the following two definitions:

```
square1 :: Integer -> Integer
square1 x = x^2

square2 :: Integer -> Integer
square2 = \ x -> x^2
```

In the first of these, the function is defined by means of an unguarded equation. In the second, the function is defined as a lambda abstract. The Haskell way of lambda abstraction goes like this. The syntax is: `\ v -> body`, where `v` is a variable of the argument type and `body` an expression of the result type. It is allowed to abbreviate `\ v -> \ w -> body` to `\ v w -> body`. And so on, for more than two variables. E.g., both of the following are correct:

```
m1 :: Integer -> Integer -> Integer
m1 = \ x -> \ y -> x*y

m2 :: Integer -> Integer -> Integer
m2 = \ x y -> x*y
```

And again, the choice of variables does not matter.

Also, it is possible to abstract over tuples. Compare the following definition of a function that solves quadratic equations by means of the well-known 'abc'-formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

```
solveQdr :: (Float,Float,Float) -> (Float,Float)
solveQdr =  \ (a,b,c) -> if a == 0 then error "not quadratic"
                         else let d = b^2 - 4*a*c in
                         if d < 0 then error "no real solutions"
                         else
                           ((- b + sqrt d) / 2*a,
                            (- b - sqrt d) / 2*a)
```

To solve the equation $x^2 - x - 1 = 0$, use `solveQdr (1,-1,-1)`, and you will get the (approximately correct) answer `(1.61803,-0.618034)`. Approximately correct, for 1.61803 is an approximation of the golden ratio, $\frac{1+\sqrt{5}}{2}$, and -0.618034 is an approximation of $\frac{1-\sqrt{5}}{2}$.

One way to think about quantified expressions like $\forall xPx$ and $\exists yPy$ is as combinations of a quantifier expression $\forall$ or $\exists$ and a lambda term $\lambda x.Px$ or $\lambda y.Py$. The lambda abstract $\lambda x.Px$ denotes the property of being a P. The quantifier $\forall$ is a function that maps properties to truth values according to the recipe: if the property holds of the whole domain then **t**, else **f**. The quantifier $\exists$ is a function that maps properties to truth values according to the recipe: if the property holds of anything at all then **t**, else **f**. This perspective on quantification is the basis of the Haskell implementation of quantifiers in Section 2.8.

2.5 Definitions and Implementations

Here is an example of a definition in mathematics. A natural number n is **prime** if $n > 1$ and no number m with $1 < m < n$ divides n.

We can capture this definition of being prime in a formula, using $m|n$ for 'm divides n', as follows (we assume the natural numbers as our domain of discourse):

$$n > 1 \wedge \neg\exists m(1 < m < n \wedge m|n). \tag{2.4}$$

Another way of expressing this is the following:

$$n > 1 \wedge \forall m((1 < m < n) \Rightarrow \neg m|n). \tag{2.5}$$

If you have trouble seeing that formulas (2.4) and (2.5) mean the same, don't worry. We will study such *equivalences* between formulas in the course of this chapter.

If we take the domain of discourse to be the domain of the natural numbers $\mathbb{N} = \{0,1,2,\ldots\}$, then formula (2.5) expresses that n is a prime number.

We can make the fact that the formula is meant as a definition explicit by introducing a predicate name P and linking that to the formula:[4]

$$P(n) \;:\equiv\; n > 1 \wedge \forall m((1 < m < n) \Rightarrow \neg m|n). \tag{2.6}$$

One way to think about this definition is as a **procedure** for testing whether a natural number is prime. Is 83 a prime? Yes, because none of $2,3,4,\ldots,9$ divides 83. Note that there is no reason to check $10,\ldots$, for since $10 \times 10 > 83$ any factor m of 83 with $m \geqslant 10$ will not be the smallest factor of 83, and a smaller factor should have turned up before.

The example shows that we can make the prime procedure more efficient. We only have to try and find the *smallest* factor of n, and any b with $b^2 > n$ cannot be the smallest factor. For suppose that a number b with $b^2 \geqslant n$ divides n. Then there is a number a with $a \times b = n$, and therefore $a^2 \leqslant n$, and a divides n. Our definition can therefore run:

$$P(n) \;:\equiv\; n > 1 \wedge \forall m((1 < m \wedge m^2 \leqslant n) \Rightarrow \neg m|n). \tag{2.7}$$

In Chapter 1 we have seen that this definition is equivalent to the following:

$$P(n) \;:\equiv\; n > 1 \wedge \mathrm{LD}(n) = n. \tag{2.8}$$

The Haskell implementation of the primality test was given in Chapter 1.

[4] $:\equiv$ means: 'is by definition equivalent to'.

2.6 Abstract Formulas and Concrete Structures

The formulas of Section 2.1 are "handled" using truth values and tables. Quantificational formulas need a **structure** to become meaningful. Logical sentences involving variables can be interpreted in quite different structures. A structure is a domain of quantification, together with a meaning for the abstract symbols that occur. A meaningful statement is the result of interpreting a logical formula in a certain structure. It may well occur that interpreting a given formula in one structure yields a true statement, while interpreting the same formula in a different structure yields a false statement. This illustrates the fact that we can use one logical formula for many different purposes.

Look again at the example formula (2.3), now displayed without reference to $\mathbb{Q}$ and using a neutral symbol **R**. This gives:

$$\forall x \forall y \, (x\mathbf{R}y \implies \exists z \, (x\mathbf{R}z \wedge z\mathbf{R}y)). \tag{2.9}$$

It is only possible to read this as a meaningful statement if

1. it is understood which is the underlying domain of quantification,

   and

2. what the symbol **R** stands for.

Earlier, the set of rationals $\mathbb{Q}$ was used as the domain, and the ordering $<$ was employed instead of the —in itself meaningless— symbol **R**. In the context of $\mathbb{Q}$ and $<$, the quantifiers $\forall x$ and $\exists z$ in (2.9) should be read as: *for all rationals* x ..., resp., *for some rational* z ..., whereas **R** should be viewed as standing for $<$. In that particular case, the formula expresses the *true* statement that, between every two rationals, there is a third one.

However, one can also choose the set $\mathbb{N} = \{0,1,2,\ldots\}$ of natural numbers as domain and the corresponding ordering $<$ as the meaning of **R**. In that case, the formula expresses the *false* statement that between every two natural numbers there is a third one.

A specification of (i) a domain of quantification, to make an unrestricted use of the quantifiers meaningful, and (ii) a meaning for the unspecified symbols that may occur (here: **R**), will be called a *context* or a *structure* for a given formula.

As you have seen here: given such a context, the formula can be "read" as a meaningful assertion about this context that can be either true or false.

Open Formulas, Free Variables, and Satisfaction. If one deletes the first quantifier expression $\forall x$ from the example formula (2.9), then the following remains:

$$\forall y\,(x\mathbf{R}y \implies \exists z\,(x\mathbf{R}z \wedge z\mathbf{R}y)). \tag{2.10}$$

Although this expression does have the *form* of a statement, it in fact is not such a thing. Reason: statements are either *true* or *false*; and, even if a quantifier domain and a meaning for **R** were specified, what results cannot be said to be true or false, as long as we do not know what it is that the variable x (which no longer is bound by the quantifier $\forall x$) stands for.

However, the expression can be turned into a statement again by replacing the variable x by (the name of) some object in the domain, or —what amounts to the same— by agreeing that x denotes this object.

For instance, if the domain consists of the set $\mathbb{N} \cup \{q \in \mathbb{Q} \mid 0 < q < 1\}$ of natural numbers together with all rationals between 0 and 1, and the meaning of **R** is the usual ordering relation $<$ for these objects, then the expression turns into a truth upon replacing x by 0.5 or by assigning x this value. We say that 0.5 *satisfies* the formula in the given domain.

However, (2.10) turns into a falsity when we assign 2 to x; in other words, 2 *does not* satisfy the formula.

Of course, one can delete a next quantifier as well, obtaining:

$$x\mathbf{R}y \implies \exists z\,(x\mathbf{R}z \wedge z\mathbf{R}y).$$

Now, both x and y have become free, and, next to a context, values have to be assigned to both these variables in order to determine a truth value.

An occurrence of a variable in an expression that is not (any more) in the scope of a quantifier is said to be *free* in that expression. Formulas that contain free variables are called *open*.

An open formula can be turned into a statement in two ways: (i) adding quantifiers that bind the free variables; (ii) replacing the free variables by (names of) objects in the domain (or stipulating that they have such objects as values).

Exercise 2.37 Consider the following formulas.

1. $\forall x \forall y (x\mathbf{R}y)$,
2. $\forall x \exists y (x\mathbf{R}y)$.
3. $\exists x \forall y (x\mathbf{R}y)$.
4. $\exists x \forall y (x = y \vee x\mathbf{R}y)$.

5. $\forall x \exists y(x\mathbf{R}y \wedge \neg\exists z(x\mathbf{R}z \wedge z\mathbf{R}y))$.

Are these formulas *true* or *false* in the following contexts?:

a. Domain: $\mathbb{N} = \{0,1,2,\ldots\}$; meaning of **R**: $<$,

b. Domain: $\mathbb{N}$; meaning of **R**: $>$,

c. Domain: $\mathbb{Q}$ (the set of rationals); meaning of **R**: $<$,

d. Domain: $\mathbb{R}$ (the set of reals); meaning of $x\mathbf{R}y$: $y^2 = x$,

e. Domain: set of all human beings; meaning of **R**: father-of,

f. Domain: set of all human beings; meaning of $x\mathbf{R}y$: x loves y.

Exercise 2.38 In Exercise 2.37, delete the first quantifier on x in formulas 1–5. Determine for which values of x the resulting open formulas are satisfied in each of the structures a–f.

2.7 Logical Handling of the Quantifiers

Goal To learn how to recognize simple logical equivalents involving quantifiers, and how to manipulate negations in quantified contexts.

Validities and Equivalents. Compare the corresponding definitions in Section 2.2.

1. A logical formula is called (logically) *valid* if it turns out to be true in *every* structure.

2. Formulas are (logically) *equivalent* if they obtain the same truth value in *every* structure (i.e., if there is no structure in which one of them is true and the other one is false).

   Notation: $\Phi \equiv \Psi$ expresses that the quantificational formulas Φ and Ψ are equivalent.

Exercise 2.39 (The propositional version of this is in Exercise 2.19 p. 49.) Argue that Φ and Ψ are equivalent iff $\Phi \Leftrightarrow \Psi$ is valid.

Because of the reference to *every possible* structure (of which there are infinitely many), these are quite complicated definitions, and it is nowhere suggested that you will be expected to decide on validity or equivalence in every case that you may encounter. In fact, in 1936 it was proved rigorously, by Alonzo Church (1903–1995) and Alan Turing (1912–1954) that no one can! This illustrates that the complexity of quantifiers exceeds that of the logic of connectives, where truth tables allow you to decide on such things in a mechanical way, as is witnessed by the Haskell functions that implement the equivalence checks for propositional logic.

Nevertheless: the next theorem already shows that it is sometimes very well possible to recognize whether formulas are valid or equivalent — if only these formulas are sufficiently simple.

Only a few useful equivalents are listed next. Here, $\Psi(x)$, $\Phi(x,y)$ and the like denote logical formulas that may contain variables x (or x,y) free.

Theorem 2.40

1. $\forall x \forall y \Phi(x,y) \equiv \forall y \forall x \Phi(x,y)$;
   $\exists x \exists y \Phi(x,y) \equiv \exists y \exists x \Phi(x,y)$,
2. $\neg \forall x \Phi(x) \equiv \exists x \neg \Phi(x)$;
   $\neg \exists x \Phi(x) \equiv \forall x \neg \Phi(x)$;
   $\neg \forall x \neg \Phi(x) \equiv \exists x \Phi(x)$;
   $\neg \exists x \neg \Phi(x) \equiv \forall x \Phi(x)$,
3. $\forall x(\Phi(x) \wedge \Psi(x)) \equiv (\forall x \Phi(x) \wedge \forall x \Psi(x))$;
   $\exists x(\Phi(x) \vee \Psi(x)) \equiv (\exists x \Phi(x) \vee \exists x \Psi(x))$.

Proof. There is no neat truth table method for quantification, and there is no neat proof here. You just have to follow common sense. For instance (part 2, first item) common sense dictates that not every x satisfies Φ if, and only if, some x does not satisfy Φ. ∎

Of course, common sense may turn out not a good adviser when things get less simple. Chapter 3 hopefully will (partly) resolve this problem for you.

Exercise 2.41 For every sentence Φ in Exercise 2.36 (p. 57), consider its negation $\neg\Phi$, and produce a more positive equivalent for $\neg\Phi$ by working the negation symbol through the quantifiers.

Order of Quantifiers. Theorem 2.40.1 says that the order of similar quantifiers (all universal or all existential) is irrelevant. But note that this is not the case for quantifiers of different kind.

On the one hand, if you know that $\exists y \forall x \Phi(x,y)$ (which states that there is one y such that for all x, $\Phi(x,y)$ holds) is true in a certain structure, then *a fortiori* $\forall x \exists y \Phi(x,y)$ will be true as well (for each x, take this *same* y). However, if $\forall x \exists y \Phi(x,y)$ holds, it is far from sure that $\exists y \forall x \Phi(x,y)$ holds as well.

Example 2.42 The statement that $\forall x \exists y (x < y)$ is true in $\mathbb{N}$, but the statement $\exists y \forall x (x < y)$ in this structure wrongly asserts that there exists a greatest natural number.

Restricted Quantification. You have met the use of restricted quantifiers, where the restriction on the quantified variable is membership in some domain. But there are also other types of restriction.

Example 2.43 (Continuity) According to the "ε-δ-definition" of continuity, a real function f is *continuous* if (domain $\mathbb{R}$):

$$\forall x \forall \varepsilon > 0 \exists \delta > 0 \forall y (|x-y| < \delta \implies |f(x)-f(y)| < \varepsilon).$$

This formula uses the restricted quantifiers $\forall \varepsilon > 0$ and $\exists \delta > 0$ that enable a more compact formulation here.

Example 2.44 Consider our example statement (2.3). Here it is again:

$$\forall y \forall x (x < y \implies \exists z (x < z \wedge z < y))$$

This can also be given as

$$\forall y \forall x < y \exists z < y (x < z),$$

but this reformulation stretches the use of this type of restricted quantification probably a bit too much.

Remark. If A is a subset of the domain of quantification, then

$\forall x \in A \, \Phi(x)$ means the same as $\forall x (x \in A \Rightarrow \Phi(x))$,

whereas

$\exists x \in A \, \Phi(x)$ is tantamount with $\exists x (x \in A \wedge \Phi(x))$.

Warning: The restricted *universal* quantifier is explained using $\Rightarrow$, whereas the *existential* quantifier is explained using $\wedge$!

Example 2.45 'Some Mersenne numbers are prime' is correctly translated as $\exists x(Mx \wedge Px)$. The translation $\exists x(Mx \Rightarrow Px)$ is wrong. It is much too weak, for it expresses (in the domain $\mathbb{N}$) that there is a natural number x which is either not a Mersenne number or it is a prime. Any prime will do as an example of this, and so will any number which is not a Mersenne number.

In the same way, 'all prime numbers have irrational square roots' is translated as $\forall x \in \mathbb{R}(Px \Rightarrow \sqrt{x} \notin \mathbb{Q})$. The translation $\forall x \in \mathbb{R}(Px \wedge \sqrt{x} \notin \mathbb{Q})$ is wrong. This time we end up with something which is too strong, for this expresses that every real number is a prime number with an irrational square root.

Restricted Quantifiers Explained. There is a version of Theorem 2.40 that employs restricted quantification. This version states, for instance, that $\neg\forall x{\in}A\,\Phi$ is equivalent to $\exists x{\in}A\,\neg\Phi$, and so on. The equivalence follows immediately from the remark above. We now have, e.g., that $\neg\forall x{\in}A\Phi(x)$ is equivalent to $\neg\forall x(x \in A \Rightarrow \Phi(x))$, which in turn is equivalent to (Theorem 2.40) $\exists x\neg(x \in A \Rightarrow \Phi(x))$, hence to (and *here* the implication turns into a conjunction — cf. Theorem 2.10) $\exists x(x \in A \wedge \neg\Phi(x))$, and, finally, to $\exists x{\in}A\neg\Phi(x)$.

Exercise 2.46 Does it hold that $\neg\exists x{\in}A\,\Phi(x)$ is equivalent to $\exists x{\notin}A\,\Phi(x)$? If your answer is 'yes', give a proof, if 'no', then you should show this by giving a simple refutation (an example of formulas and structures where the two formulas have different truth values).

Exercise 2.47 Is $\exists x{\notin}A\,\neg\Phi(x)$ equivalent to $\exists x{\in}A\,\neg\Phi(x)$? Give a proof if your answer is 'yes', and a refutation otherwise.

Exercise 2.48 Produce the version of Theorem 2.40 (p. 64) that employs restricted quantification. Argue that your version is correct.

Example 2.49 (Discontinuity Explained) The following formula describes what it means for a real function f to be discontinuous in x:

$$\neg\,\forall\varepsilon>0\,\exists\delta>0\,\forall y\,(\,|x-y|<\delta \implies |f(x)-f(y)|<\varepsilon\,).$$

Using Theorem 2.40, this can be transformed in three steps, moving the negation over the quantifiers, into:

$$\exists\varepsilon>0\,\forall\delta>0\,\exists y\,\neg\,(\,|x-y|<\delta \implies |f(x)-f(y)|<\varepsilon\,).$$

According to Theorem 2.10 this is equivalent to

$$\exists\varepsilon>0\,\forall\delta>0\,\exists y\,(\,|x-y|<\delta \wedge \neg\,|f(x)-f(y)|<\varepsilon\,),$$

i.e., to

$$\exists\varepsilon>0\,\forall\delta>0\,\exists y\,(\,|x-y|<\delta \wedge |f(x)-f(y)|\geqslant\varepsilon\,).$$

What has emerged now is a clearer "picture" of what it means to be discontinuous in x: there must be an $\varepsilon>0$ such that for every $\delta>0$ ("no matter how small") a y can be found with $|x-y|<\delta$, whereas $|f(x)-f(y)|\geqslant\varepsilon$; i.e., there are numbers y "arbitrarily close to x" such that the values $f(x)$ and $f(y)$ remain at least ε apart.

Different Sorts. Several *sorts* of objects, may occur in one and the same context. (For instance, sometimes a problem involves vectors as well as reals.) In such a situation, one often uses different variable naming conventions to keep track of the differences between the sorts. In fact, sorts are just like the basic types in a functional programming language.

Just as good naming conventions can make a program easier to understand, naming conventions can be helpful in mathematical writing. For instance: the letters $n,m,k,\ldots$ are often used for natural numbers, $f,g,h,\ldots$ usually indicate that functions are meant, etc.

The interpretation of quantifiers in such a case requires that not one, but several domains are specified: one for every sort or type. Again, this is similar to providing explicit typing information in a functional program for easier human digestion.

Exercise 2.50 That the sequence $a_0,a_1,a_2,\ldots\in\mathbb{R}$ converges to a, i.e., that $\lim_{n\to\infty}a_n=a$, means that $\forall\delta>0\exists n\forall m\geqslant n(|a-a_m|<\delta)$. Give a positive equivalent for the statement that the sequence $a_0,a_1,a_2,\ldots\in\mathbb{R}$ does not converge.

2.8 Quantifiers as Procedures

One way to look at the meaning of the universal quantifier $\forall$ is as a procedure to test whether a set has a certain property. The test yields **t** if the set equals the whole domain of discourse, and **f** otherwise. This means that $\forall$ is a procedure that maps the domain of discourse to **t** and all other sets to **f**. Similarly for restricted universal quantification. A restricted universal quantifier can be viewed as a procedure that takes a set A and a property P, and yields **t** just in case the set of members of A that satisfy P equals A itself.

In the same way, the meaning of the unrestricted existential quantifier $\exists$ can be specified as a procedure. $\exists$ takes a set as argument, and yields **t** just in case the argument set is non-empty. A restricted existential quantifier can be viewed as a procedure that takes a set A and a property P, and yields **t** just in case the set of members of A that satisfy P is non-empty.

If we implement sets as lists, it is straightforward to implement these quantifier procedures. In Haskell, they are predefined as `all` and `any` (these definitions will be explained below):

```
any, all       :: (a -> Bool) -> [a] -> Bool
any p           = or  . map p
all p           = and . map p
```

The typing we can understand right away. The functions `any` and `all` take as their first argument a function with type `a` inputs and type `Bool` outputs (i.e., a test for a property), as their second argument a list over type `a`, and return a truth value. Note that the list representing the restriction is the *second* argument.

To understand the implementations of `all` and `any`, one has to know that `or` and `and` are the generalizations of (inclusive) disjunction and conjunction to lists. (We have already encountered `and` in Section 2.2.) They have type `[Bool] -> Bool`.

Saying that all elements of a list `xs` satisfy a property `p` boils down to: the list `map p xs` contains only `True` (see Section 1.8). Similarly, saying that some element of a list `xs` satisfies a property `p` boils down to: the list `map p xs` contains at least one `True`. This explains the implementation of `all`: first apply `map p`, next apply `and`. In the case of `any`: first apply `map p`, next apply `or`.

The action of applying a function `g :: b -> c` after a function `f :: a -> b` is performed by the function `g . f :: a -> c` , the composition of `f` and `g`. See Section 6.3 below.

The definitions of `all` and `any` are used as follows:

```
Prelude> any (<3) [0..]
True
Prelude> all (<3) [0..]
False
Prelude>
```

The functions `every` and `some` get us even closer to standard logical notation. These functions are like `all` and `any`, but they first take the restriction argument, next the body:

```
every, some :: [a] -> (a -> Bool) -> Bool
every xs p = all p xs
some  xs p = any p xs
```

Now, e.g., the formula $\forall x \in \{1,4,9\} \exists y \in \{1,2,3\}\ x = y^2$ can be implemented as a test, as follows:

```
TAMO> every [1,4,9] (\ x -> some [1,2,3] (\ y -> x == y^2))
True
```

But caution: the implementations of the quantifiers are procedures, not algorithms. A call to `all` or `any` (or `every` or `some`) need not terminate. The call

```
every [0..] (>=0)
```

will run forever. This illustrates once more that the quantifiers are in essence more complex than the propositional connectives. It also motivates the development of the method of proof, in the next chapter.

Exercise 2.51 Define a function `unique :: (a -> Bool) -> [a] -> Bool` that gives `True` for `unique p xs` just in case there is exactly one object among `xs` that satisfies `p`.

Exercise 2.52 Define a function `parity :: [Bool] -> Bool` that gives `True` for `parity xs` just in case an even number of the `xss` equals `True`.

Exercise 2.53 Define a function `evenNR :: (a -> Bool) -> [a] -> Bool` that gives `True` for `evenNR p xs` just in case an even number of the `xss` have property `p`. (Use the `parity` function from the previous exercise.)

2.9 Further Reading

If you find that the pace of the introduction to logic in this chapter is too fast for you, you might wish to have a look at the more leisurely paced [NK04]. Excellent books about computer science applications of logic are [Bur98] and [HR00]. Good introductions to mathematical logic are Ebbinghaus, Flum and Thomas [EFT94], and Chiswell and Hodges [CH07].