

The Haskell School of Expression

LEARNING FUNCTIONAL PROGRAMMING
THROUGH MULTIMEDIA

PAUL HUDAK

THE HASKELL SCHOOL OF EXPRESSION

Functional programming is a programming style that emphasizes the use of functions (in contrast to object-oriented programming, which emphasizes the use of objects). It has become popular in recent years because of its simplicity, conciseness, and clarity. This book teaches functional programming using Haskell, the most popular purely functional language. The emphasis is on functional programming as a way of thinking and problem solving, using Haskell as a vehicle for expressing solutions. Rather than using conventional - arguably boring - examples from mathematics, which are commonly found in other programming language books, this tutorial uses examples drawn from multimedia applications, including graphics, animation, and computer music, thus rewarding the reader with working programs for inherently more interesting applications. The author also teaches how to reason about functional programs, using a very simple process of calculation.

Aimed at both beginning and advanced programmers, this tutorial begins with a gentle introduction to functional programming, including basic ideas such as values, types, pattern matching, recursion, higher-order functions, data structures, polymorphism, abstraction, lazy evaluation, and proof by calculation. It then moves rapidly on to more advanced topics, such as infinite data structures, type classes, higher-order types, IO, monads, and inductive proofs. Details about programming in Haskell are presented in boxes throughout the text for easy reference.

Paul Hudak is Professor of Computer Science at Yale University. He was instrumental in organizing and chairing the Haskell Committee, an international group of computer scientists who designed Haskell, a nonstrict, pure functional programming language. He is a cofounder and editor of the *Journal of Functional Programming*. He has published over 100 papers on the design and application of programming languages.

Hudak believes that programming languages should be pushed further in the direction of high-level abstractions, in which the programmer says less about the details of a computation and more about the problem specification itself. At the same time, he recognizes the need for smart compilation techniques to make such languages practical. His most recent interest is in applying these principles to multimedia technology, including computer music, graphics and animation, and robotics.

The Haskell School of Expression

LEARNING FUNCTIONAL PROGRAMMING
THROUGH MULTIMEDIA

PAUL HUDAK

Yale University



PUBLISHED BY THE PRESS SYNDICATE OF THE UNIVERSITY OF CAMBRIDGE
The Pitt Building, Trumpington Street, Cambridge, United Kingdom

CAMBRIDGE UNIVERSITY PRESS

The Edinburgh Building, Cambridge CB2 2RU, UK <http://www.cup.cam.ac.uk>
40 West 20th Street, New York, NY 10011-4211, USA <http://www.cup.org>
10 Stamford Road, Oakleigh, Melbourne 3166, Australia
Ruiz de Alarcón 13, 28014 Madrid, Spain

www.cambridge.org

Information on this title: www.cambridge.org/9780521643382

© Cambridge University Press 2000

This book is in copyright. Subject to statutory exception
and to the provisions of relevant collective licensing agreements,
no reproduction of any part may take place without
the written permission of Cambridge University Press.

First published 2000

Printed in the United States of America

Typeface Lucida Y&Y 9.25/13 pt. and Optima *System* L^AT_EX 2_ε [TB]

A catalog record for this book is available from the British Library.

Library of Congress Cataloging in Publication Data

Hudak, Paul.

The Haskell school of expression: learning functional programming through
multimedia / Paul Hudak

p. cm.

ISBN 0-521-64338-4 (hardback) – ISBN 0-521-64408-9 (pbk.)

1. Functional programming (Computer science) 2. Multimedia systems. I. Title.

QA76.62 H83 2000

005.1'14 21 – dc21

99-045529

ISBN 0 521 64338 4 hardback

ISBN 0 521 64408 9 paperback

*This book is dedicated to
Cathy, Cristina, Jennifer, and Rusty*

Contents

<i>Preface</i>	<i>page</i>	<i>xiii</i>
1 Problem Solving, Programming, and Calculation		1
1.1 Computation by Calculation in Haskell		2
1.2 Expressions, Values, and Types		6
1.3 Function Types and Type Signatures		8
1.4 Abstraction, Abstraction, Abstraction		10
1.4.1 Naming		10
1.4.2 Functional Abstraction		12
1.4.3 Data Abstraction		13
1.5 Code Reuse and Modularity		17
1.6 Beware of Programming with Numbers		17
2 A Module of Shapes: Part I		21
2.1 Geometric Shapes		22
2.2 Areas of Shapes		25
2.3 Cleaning Up		34
3 Simple Graphics		35
3.1 Basic Input/Output		36
3.2 Graphics Windows		40
3.3 Drawing Graphics Other Than Text		42
3.4 Some Examples		43
4 Shapes II: Drawing Shapes		48
4.1 Dealing With Different Coordinate Systems		48
4.2 Converting Shapes to Graphics		51
4.3 Some Examples		52
4.4 In Retrospect		54
5 Polymorphic and Higher-Order Functions		56
5.1 Polymorphic Types		56

5.2	Abstraction Over Recursive Definitions	58
5.2.1	Map is Polymorphic	59
5.2.2	Using <i>map</i>	60
5.3	Append	63
5.3.1	The Efficiency and Fixity of Append	64
5.4	Fold	65
5.4.1	Haskell's Folds	67
5.4.2	Why Two Folds?	68
5.5	A Final Example: Reverse	69
5.6	Errors	71
6	Shapes III: Perimeters of Shapes	74
6.1	Perimeters of Shapes	74
7	Trees	81
7.1	A Tree Data Type	81
7.2	Operations on Trees	83
7.3	Arithmetic Expressions	84
8	A Module of Regions	87
8.1	The Region Data Type	87
8.2	The Meaning of Shapes and Regions	91
8.2.1	The Meaning of Shapes	92
8.2.2	The Encoding of the Meaning of Shapes	93
8.2.3	The Meaning of Regions	96
8.2.4	The Encoding of the Meaning of Regions	97
8.3	Algebraic Properties of Regions	99
8.4	In Retrospect	101
9	More About Higher-Order Functions	105
9.1	Currying	105
9.2	Sections	108
9.3	Anonymous Functions	110
9.4	Function Composition	111
10	Drawing Regions	114
10.1	The Picture Data Type	115
10.2	Drawing Pictures	115
10.3	Drawing Regions	116
10.3.1	From Regions to Graphics Regions: First Attempt	117
10.3.2	From Regions to Graphics Regions: Second Attempt	119
10.3.3	Translating Shapes into Graphics Regions	121
10.3.4	Examples	123

10.4	User Interaction	126
10.5	Putting it all Together	127
10.5.1	Examples	128
11	Proof by Induction	131
11.1	Induction and Recursion	131
11.2	Examples of List Induction	132
11.2.1	Proving Function Equivalences	133
11.3	Useful Properties on Lists	137
11.3.1	Function Strictness	137
11.4	Induction on Other Data Types	141
11.4.1	A More Efficient Exponentiation Function	143
12	Qualified Types	147
12.1	Equality	148
12.2	Defining Your Own Type Classes	150
12.3	Inheritance	153
12.4	Haskell's Standard Type Classes	154
12.5	Derived Instances	157
12.6	Reasoning With Type Classes	160
13	A Module of Simple Animations	163
13.1	What is an Animation?	163
13.2	Representing an Animation	165
13.3	An Animator	167
13.4	Fun With Type Classes	172
13.4.1	Rising to the Level of Animations	172
13.4.2	Type Classes to the Rescue	172
13.4.3	Defining New Type Classes for Behaviors	176
13.5	Lifting to the Limit	177
13.6	Time Transformation	179
13.7	A Final Example: A Kaleidoscope Program	180
14	Programming With Streams	187
14.1	Lazy Evaluation	187
14.2	Recursive Streams	190
14.3	Stream Diagrams	193
14.4	Lazy Patterns	195
14.5	Memoization	198
14.6	Inductive Properties of Infinite Lists	201
15	A Module of Reactive Animations	208
15.1	FAL by Example	209

15.1.1	Basic Reactivity	209
15.1.2	Event Choice	210
15.1.3	Recursive Event Processing	211
15.1.4	Events with Data	212
15.1.5	Snapshot	212
15.1.6	Boolean Events	212
15.1.7	Integration	213
15.2	Implementing FAL	214
15.2.1	An Implementation Strategy	215
15.2.2	Incremental Sampling	217
15.2.3	Final Refinements	219
15.2.4	Representing Events	220
15.3	The Implementation	220
15.3.1	Behaviors	221
15.3.2	Events	225
15.3.3	An Example	228
15.4	Extensions	229
15.4.1	Variations on <i>switch</i>	231
15.4.2	Mouse Motion	232
15.5	Paddleball in Twenty Lines	233
16	Communicating With the Outside World	236
16.1	Files, Channels, and Handles	236
16.1.1	Why Use Handles?	238
16.1.2	Channels	238
16.2	Exception Handling	239
16.3	First-Class Channels and Concurrency	242
17	Rendering Reactive Animations	245
17.1	Preliminaries	245
17.2	Reactimate	246
17.3	Window User	247
18	Higher-Order Types	249
18.1	The Functor Class	249
18.2	The Monad Class	251
18.2.1	Other Instances of Monad	255
18.2.2	Other Monadic Operations	259
18.3	The MonadPlus Class	259
18.4	State Monads	261
18.5	Type Class Type Errors	263

19	An Imperative Robot Language	265
19.1	IRL by Example	266
19.2	<i>Robot</i> is a State Monad	270
19.3	The Implementation of IRL Commands	272
19.3.1	Robot Orientation	273
19.3.2	Using the Pen	274
19.3.3	Playing With Coins	274
19.3.4	Logic and Control	274
19.4	All the World is a Grid	276
19.5	Robot Graphics	281
19.6	Putting it all Together	282
20	Functional Music Composition	287
20.1	The Music Data Type	288
20.2	Higher-Level Constructions	293
20.2.1	Lines and Chords	293
20.2.2	Delay and Repeat	293
20.2.3	Polyrhythms	294
20.2.4	Determining Duration	295
20.2.5	Reversing Musical Structure	295
20.2.6	Truncating Parallel Composition	296
20.2.7	Trills	297
20.2.8	Percussion	298
20.3	A Couple of Final Examples	300
20.3.1	Cascades	300
20.3.2	Self-Similar (Fractal) Music	301
21	Interpreting Functional Music	304
21.1	Interpreting Music: A <i>Performance</i>	305
21.2	An Algebra of Music	308
22	From Performance to MIDI	313
22.1	An Introduction to MIDI	313
22.2	The Conversion Process	314
22.3	Putting It All Together	319
23	A Tour of the PreludeList Module	321
23.1	The PreludeList Module	321
23.2	Simple List Selector Functions	322
23.3	Index-Based Selector Functions	323
23.4	Predicate-Based Selector Functions	324
23.5	Fold-like Functions	325
23.6	List Generators	327

23.7	String-Based Functions	327
23.8	Boolean List Functions	328
23.9	List Membership Functions	329
23.10	Arithmetic on Lists	329
23.11	List Combining Functions	330
24	A Tour of Haskell's Standard Type Classes	332
24.1	The Ordered Class	332
24.2	The Enumeration Class	333
24.3	The Bounded Class	334
24.4	The Show Class	334
24.5	The Read Class	338
24.6	The Index Class	341
24.7	The Numeric Classes	343
A	Built-in Types Are Not Special	345
B	Pattern-Matching Details	348
	<i>Bibliography</i>	353
	<i>Index</i>	357

Preface

The first high-level languages developed for general purpose programming were *Fortran* (Backus, 1978) and *Lisp* (McCarthy, 1978), developed in the late 1950s by John Backus and John McCarthy, respectively. From Fortran grew many of today's modern *imperative* languages, most of which did not improve significantly on the fundamental ideas found in the language *Algol* (de Morgan, Hill and Wichmann, 1976), which shortly followed Fortran. The Lisp family was less fecund, perhaps because it was so far ahead of its time, but was the seed for the family of *functional* languages about which this textbook was written. The most radical in this class of languages is probably *Haskell*, originally designed in the late 1980s (Hudak, Wadler, 1988) based on, at that point, a good ten years of experience designing and implementing similar languages, most notably a series of languages developed by David Turner in the late 1970s and early 1980s (Turner, 1976; 1985). Although research continues on the design of Haskell, the version used in this textbook, *Haskell 98* (Augustsson, et al., 1999), is the latest and most stable version of the language and its libraries.

Haskell was named after the logician Haskell B. Curry who, along with Alonzo Church, established the theoretical foundations of functional programming back when computers themselves were only a gleam in researchers' eyes. A curious historical fact is that Haskell Curry's father, Samuel Silas Curry, helped to found and direct a school in Boston called the *School of Expression*.¹ Because pure functional programming centers around the notion of an *expression*, I thought that *The Haskell School of Expression* would be a good title for this book.

¹ This school eventually evolved into what is now called *Curry College* (see www.curry.edu:8080/history/history.html for a brief history of the college).

A Brief Account of Language Success Stories

It's hard to predict just how it is that a language becomes popular. Fortran became popular because it was the first high-level language and a welcome alternative to assembly language, and ultimately because it was a good vehicle for coding numerical algorithms in the domain of scientific computing. C (Kernighan and Ritchie, 1978) was not revolutionary by any means, often being described as a high-level assembly language. But it found a niche in Unix and systems-level programming, and from there to other operating system platforms. Cobol (Cobol, 1968) was designed from day one for business applications, and so made its mark. Java's (Gosling, Joy and Steele, 1996) recent astonishing rise in popularity is rather perplexing on one hand, yet quite understandable on another. As a language, it is simple and elegant, but certainly not revolutionary. Yet it found a niche in its use on the world-wide-web, which at the time was (and still is) a phenomenon in its own right. Lisp and Scheme (Rees and Clinger, 1986) for AI, Ada (Ada,79) for real-time applications, VHDL and Verilog for digital hardware design, and the list goes on.

From this discussion it appears that for a language to become popular it needs an application - a *niche* - in which it excels, or in which it at least was the first to arrive. Curiously, for most language designers, this is the *opposite* of what is desired. Most language designers would like their languages to be truly general purpose, able to solve all of the world's problems with great succinctness, clarity, and efficiency. As a result, many excellent languages out there, some with very decent implementations, will probably forever remain in obscurity because they never found their niche.

Although the designers of Haskell exhibited no exception to this ambitious approach to language design, in retrospect I wonder if there might be a plausible niche for Haskell. These sorts of things are mostly out of my control, of course, but I thought it would be fun to use *multimedia* - graphics, sound, and animation - as an underlying theme, embodied concretely through many examples and exercises, in this text. Multimedia is an application area that is certainly current and important, and one in which Haskell's advantages are highly visible.

In addition, one of the nice things about multimedia programming is that all of the really interesting (and hard!) problems faced by computer scientists over the past 30 years are found in the "virtual worlds" that we create. Nondeterminism, concurrency, state, time, efficiency, decidability, and others are all issues that must be addressed. Although I don't cover all of these issues in this text, multimedia programming nevertheless

provides a good vehicle through which one could teach general topics in computer science.

I also hope that this text demonstrates the ease with which one can embed a *domain-specific language* in Haskell. In a more general sense, this might actually be the most profitable niche for Haskell, as there have been a number of success stories in this area already.

In any case, I hope that you enjoy working through the various multimedia programs as much as I have enjoyed creating them. At the same time, I hope that this text might help Haskell to find its niche and thus avoid the fate of obscurity described earlier. Haskell really is a beautiful language.

Why I Wrote This Book, and How to Read It

At the time I began writing this book there were not many other books about programming specifically in Haskell. But that wasn't the main reason I decided to tackle this task. More importantly, there was a need for a book that described *how to solve problems using a functional language* such as Haskell. As with any major class of languages, there is a certain mind-set for contemplation, a certain viewpoint of the world, and a certain approach to problem solving that collectively work best. If you teach only Haskell language details to a C programmer, she is likely to write very ugly, incomprehensible functional programs. But if you teach her how to think differently, how to see problems in a different light, functional solutions will come easily, and elegant Haskell programs will result. That, in a nutshell, is my goal in this textbook. As Samuel Silas Curry once said:

All expression comes *from within outward*, from the center to the surface, from a hidden source to outward manifestation. The study of expression as a natural process brings you into contact with cause and makes you feel the source of reality.

(<http://www.curry.edu:8080/history/history.html>)

I encourage the seasoned programmer having experience only with conventional imperative and/or object-oriented languages to read this text with an open mind. Many things will be different, and will likely feel awkward. There will be a tendency to rely on old habits when writing new programs, and to ignore my suggestions about how to approach things differently. If you can manage to resist these tendencies, I am confident that you will have an enjoyable learning experience. Many of those who

succeed in this process find that many of the things that they learn about functional programming can be applied to imperative and object-oriented languages – after all, most of these other languages contain a significant functional subset – and that their imperative coding style changes for the better as a result.

I also ask the experienced programmer to be patient while in the earlier chapters I explain things like “syntax,” “operator precedence,” and others because my goal is that this text should be readable by someone having only modest prior programming experience. With patience the more advanced ideas will appear soon enough.

If you are a novice programmer, I suggest taking your time with the book; work through the exercises, and don’t rush things. If, however, you don’t fully grasp an idea, feel free to move on, but try to reread difficult material at a later time when you have seen more examples of the concepts in action. For the most part this is a “show by example” textbook, and you should try to execute as many of the programs in this text as you can, as well as every program that you write. Learn-by-doing is the corollary to show-by-example.

Finally, although the text begins quite gently, it moves at a fairly rapid pace, and covers many advanced ideas in functional programming, some of which are not covered in any other text that I am aware of. So there is much here even for those who are already familiar with the basics of functional programming.

Suggestions to Instructors

All of the material in this textbook can be covered in one semester as an advanced undergraduate course. For lower-level courses, including possible use in high school, some of the mathematics may cause problems, but for bright students I suspect most of the material can still be covered.

I strongly encourage sticking to the order of the chapters in the book, which introduces Haskell language features as they are demanded by the underlying application themes (generally the chapters alternate between “concepts” and “applications”). If you are an experienced functional programmer, you will see instances early in the book where a lambda expression here, or eta-conversion there, will simplify things, but I have chosen to delay such simplifications in most cases. Flooding the student with too many features early on can be overwhelming.

The only exception to following the given chapter order is that Chapters 20 to 22 provide a somewhat independent thread on computer music, and can be covered anytime after Chapter 11. The most difficult chapter

is probably Chapter 15, and the most dispensible chapters are probably Chapters 17 and 22. Also, if you want to omit nonmultimedia applications you might consider skipping Chapter 6, although that chapter contains the first introduction to infinite lists. Finally, Chapters 23 and 24 are short “tours” of the `PreludeList` Module and Standard Type Classes, respectively, and could be assigned as auxiliary reading, or covered piecemeal as related topics are introduced.

The web page <http://haskell.org/soe> contains a great deal of useful information related to the text, including libraries, source code for each chapter, PowerPoint slides, and errata. You can send email to me at paul.hudak@yale.edu with feedback, questions, corrections, etc.

Haskell Implementations

There are several good implementations of Haskell, all available free on the Internet through the Haskell home page at <http://haskell.org>. One that I especially recommend is the *Hugs* implementation, a very easy-to-use and easy-to-install Haskell interpreter. Hugs runs on a variety of platforms, including PC's (Windows 95/NT), various flavors of Unix (Linux, Solaris, HP), and Mac OS. The Glasgow Haskell Compiler (GHC) supports the same libraries as Hugs, and has the benefit of being a true compiler instead of an interpreter.

All of the code in the book is compliant with the Haskell '98 standard, and has been tested on the Hugs '98 implementation of Haskell. Unfortunately, the graphics and animation applications rely on a library that was originally developed only for Windows 95/NT. You should consult the SOE web page for the latest information regarding compatibility with other platforms.

Acknowledgments

I learned that writing a textbook is not an easy task, and could not have been accomplished without the help of many friends and colleagues. I would especially like to thank Mark Jones, with whom I started this project, and who first suggested the use of the name *School of Expression*. I'd also like to thank John Peterson and Joe Fasel for help in writing *A Gentle Introduction to Haskell* (Hudak and Fasel, 1992), from which some of this text was adapted; Alastair Reid for help with the Hugs implementation and graphics libraries; Conal Elliott for many helpful suggestions and inspirations, especially concerning graphics and animation; Erik Meijer for feedback from using a previous version of this text in his

class; Mark Tullsen for careful proofreading and tedious index generation; Tom Makucevich and John Garvin for help with computer music; Tim Sheard for feedback from using my book in teaching a functional programming course at Yale, and for creating great PowerPoint slides in the process; Zhanyong Wan for excellent feedback on the text as a Teaching Assistant in Tim's course; Sigbjorn Finne for writing the kaleidoscope program; Valery Trifonov for adapting the kaleidoscope program and other useful feedback; Linda Joyce for help with the indexing and excellent administrative support; Martin Sulzmann for help debugging graphics; Lauren Cowles, my editor, whose patience is extraordinary; and the many students in various classes at Yale who endured earlier versions of the text.

I would also like to thank the several United States funding agencies, most notably NSF and DARPA, who have provided considerable financial support for functional programming research at Yale and elsewhere.

Most of all, this work could not have been accomplished without the love and support of my family. Thank you Cathy, Cristina, and Jennifer.

Happy Haskell Hacking!

Paul Hudak
New Haven
June 1999

CHAPTER ONE

Problem Solving, Programming, and Calculation

Programming, in its broadest sense, is *problem solving*. It begins when we look out into the world and see problems that we want to solve, problems that we think can and should be solved using a digital computer. Understanding the problem well is the first – and probably the most important – step in programming, because without that understanding we may find ourselves wandering aimlessly down a dead-end alley, or worse, down a fruitless alley with no end. “Solving the wrong problem” is a phrase often heard in many contexts, and we certainly don’t want to be victims of that crime. So the first step in programming is answering the question, “What problem am I trying to solve?”

Once you understand the problem, then you must find a solution. This may not be easy, of course, and in fact you may discover several solutions, so we also need a way to measure success. There are various dimensions in which to do this, including correctness (“Will I get the right answer?”) and efficiency (“Will I have enough resources?”). But the distinction of which solution is better is not always clear, because the number of dimensions can be large, and programs will often excel in one dimension and do poorly in others. For example, there may be one solution that is fastest, one that uses the least amount of memory, and one that is easiest to understand. Choosing can be difficult and is one of the more interesting challenges that you will face in programming.

The last measure of success mentioned above – clarity of a program – is somewhat elusive, most difficult to measure, and, quite frankly, sometimes difficult to rationalize. But in large software systems clarity is an especially important goal, because the most important maxim about such systems is that they are never really finished! The process of continuing work on a software system after it is delivered to users is what software engineers call *software maintenance*, and is the most expensive phase of the so-called “software lifecycle.” Software maintenance includes fixing

bugs in programs, as well as changing certain functionality and enhancing the system with new features in response to users' experience.

Therefore, taking the time to write programs that are highly legible – easy to understand and to reason about – will facilitate the software maintenance process. It is important to realize that the person performing software maintenance is usually not the person who wrote the original program. Therefore, when you write your programs, write them as if you are writing them for someone else to see, to understand, and ultimately to pass judgment on!

In this book, I will often solve each example in several different ways (some of which are dead ends!), taking the time to contrast the style, efficiency, clarity, and functionality of the results.¹ I do this not just for pedagogical purposes. *Such reworking of programs is the norm*, and you are encouraged to get into the habit of doing so. Don't always be satisfied with your first solution to a problem, and always be prepared to go back and change parts of your program that you later discover do not satisfy your actual needs.

1.1 Computation by Calculation in Haskell

Discussions of program clarity bring us ultimately to the issue of our programming language choice. This choice determines how we express our solutions in such a way that a computer can understand them. Our programs embody our solutions – and our creativity, eloquence, and perseverance – for interpretation by the computer.

In this text I will use the programming language *Haskell* to address many of the issues discussed in the last section.² I have tried to avoid the approach of explaining Haskell first and giving examples second. Rather, I will walk with you, step by step, along the path of understanding an application, understanding the solution space, and understanding how to express a particular solution in Haskell. I want you to learn how to problem solve!

Along this path I will use whatever tools are appropriate for analyzing a particular problem domain, very often mathematical tools familiar to the average college student, indeed most to the average high school student. Concurrently, I will evolve our problems toward a particular view of computation that is especially useful: that of *computation by*

¹ At times I also explore different methods for proving properties of programs.

² If this were a text on software engineering, I would address many other issues as well. The methods that I describe are consistent with the principles of software engineering, but detailed discussion of those principles is beyond the scope of this textbook.

calculation. You will find that such a viewpoint is not only powerful, it is also *simple* (we won't shy away from difficult problems). Haskell supports well the idea of computation by calculation. Programs in Haskell can be viewed as *functions* whose input is that of the problem being solved, and whose output is our desired result; and the behavior of functions can be understood easily as computation by calculation.

An example might help to demonstrate these ideas. Suppose we want to perform an arithmetic calculation such as $3 \times (9 + 5)$. In Haskell we would write this as $3 * (9 + 5)$ because most standard computer keyboards and text editors do not recognize the special symbol \times . To calculate the result, we proceed as follows:

$$\begin{aligned} 3 * (9 + 5) \\ \Rightarrow 3 * 14 \\ \Rightarrow 42 \end{aligned}$$

It turns out that this is not the only way to compute the result, as evidenced by this alternative calculation:³

$$\begin{aligned} 3 * (9 + 5) \\ \Rightarrow 3 * 9 + 3 * 5 \\ \Rightarrow 27 + 3 * 5 \\ \Rightarrow 27 + 15 \\ \Rightarrow 42 \end{aligned}$$

Even though this calculation takes two extra steps, it at least gives the correct answer. Indeed, an important property of each and every program in this textbook – in fact every program that can be written in the functional language Haskell – is that it will always yield the same answer when given the same inputs, regardless of the order in which we choose to perform the calculations.⁴ This is precisely the mathematical definition of a function: For the same inputs, it always yields the same output.

On the other hand, the first calculation above took fewer steps than the second, and so we say that it is more *efficient*. Efficiency in both space (amount of memory used) and time (number of steps executed) is important when searching for solutions to problems, but of course if we get the wrong answer, efficiency is a moot point. In general, we will search first for any solution to a problem, and later refine it for better performance.

³ This assumes that multiplication distributes over addition in the number system being used, a point that I will return to later.

⁴ As long as we don't choose a nonterminating sequence of calculations, another issue that we will return to later.

The above calculations are fairly trivial, of course. But we will be doing much more sophisticated operations soon enough. For starters - and to introduce the idea of a function - we could *generalize* the arithmetic operations performed in the previous example by defining a function to perform them for any numbers x , y , and z :

$$\text{simple } x \ y \ z = x * (y + z)$$

This equation defines *simple* as a function of three *arguments*, x , y , and z . In mathematical notation, we might see the above written slightly differently, namely:

$$\text{simple}(x, y, z) = x \times (y + z)$$

In any case, it should be clear that "*simple* 3 9 5" is the same as " $3 * (9 + 5)$." In fact the proper way to calculate the result is:

$$\begin{aligned} \text{simple } 3 \ 9 \ 5 \\ \Rightarrow 3 * (9 + 5) \\ \Rightarrow 3 * 14 \\ \Rightarrow 42 \end{aligned}$$

The first step in this calculation is an example of *unfolding* a function definition: 3 is substituted for x , 9 for y , and 5 for z on the right-hand side of the definition of *simple*. This is an entirely mechanical process, not unlike what the computer actually does to execute the program.

When I wish to say that an expression e evaluates (via zero, one, or possibly many more steps) to the value v , I will write $e \Rightarrow v$ (this arrow is longer than that used earlier). So we can say directly, for example, that *simple* 3 9 5 \Rightarrow 42, which should be read "*simple* 3 9 5 evaluates to 42."

With *simple* now suitably defined, we can repeat the sequence of arithmetic calculations as often as we like, using different values for the arguments to *simple*. For example, *simple* 4 3 2 \Rightarrow 20.

We can also use calculation to *prove properties* about programs. For example, it should be clear that for any a , b , and c , *simple* $a \ b \ c$ should yield the same result as *simple* $a \ c \ b$. For a proof of this, we calculate *symbolically*; that is, using the symbols a , b , and c rather than concrete numbers such as 3, 5, and 9:

$$\begin{aligned} \text{simple } a \ b \ c \\ \Rightarrow a * (b + c) \\ \Rightarrow a * (c + b) \\ \Rightarrow \text{simple } a \ c \ b \end{aligned}$$

The same notation will be used for these symbolic steps as for concrete ones. In particular, the arrow in the notation reflects the direction of our reasoning, and nothing more. In general, if $e1 \Rightarrow e2$, then it's also true that $e2 \Rightarrow e1$.

I will also refer to these symbolic steps as “calculations,” even though the computer will not typically perform them when executing a program (although it might perform them *before* a program is run if it thinks that it might make the program run faster). The second step in the calculation above relies on the commutativity of addition (namely that, for any numbers x and y , $x + y = y + x$). The third step is the reverse of an unfold step, and is appropriately called a *fold* calculation. It would be particularly strange if a computer performed this step while executing a program, because it does not seem to be headed toward a final answer. But for proving properties about programs, such “backward reasoning” is quite important.

When I wish to make the justification for each step clearer, whether symbolic or concrete, a calculation will be presented with more detail, as in:

```

simple a b c
  ⇒ { unfold }
a * (b + c)
  ⇒ { commutativity }
a * (c + b)
  ⇒ { fold }
simple a c b

```

In most cases, however, this will not be necessary.

Proving properties of programs is another theme that will be repeated often in this text. As the world relies more and more on computers to accomplish not just ordinary tasks such as writing term papers and sending email, but also life-critical tasks such as controlling medical procedures and guiding spacecraft, then the correctness of programs gains in importance. Proving complex properties of large, complex programs is not easy, and is rarely if ever done in practice. However, that should not deter us from proving simpler properties of the whole system, or complex properties of parts of the system, because such proofs may uncover errors, and if not, at least help us to gain confidence in our effort.

If you are already an experienced programmer, the idea of computing *everything* by calculation may seem odd at best and naive at worst. How does one write to a file, draw a picture, or respond to mouse clicks? If you are wondering about these things, have patience reading the early chapters and find delight reading the later chapters where the full power

of this approach begins to shine. I will avoid, however, most comparisons between Haskell and conventional programming languages such as C, C++, Ada, Java, or even Scheme or ML (two “almost functional” languages), because for those who have programmed in these other languages the differences will be obvious, and for those who haven’t the comments would be superfluous.

In many ways this first chapter is the most difficult chapter in the entire text because it contains the highest density of new concepts. If you have trouble with some of the ideas here, keep in mind that we will return to almost every idea at later points in the text. And don’t hesitate to return to this chapter later to reread difficult sections; they will likely be much easier to grasp at that time.

Exercise 1.1 Write out all of the steps in the calculation of the value of *simple (simple 2 3 4) 5 6*

Exercise 1.2 Prove by calculation that $\text{simple } (a - b) a b \Rightarrow a^2 - b^2$.

DETAILS

In this text the need will often arise to explain some aspect of Haskell in more detail, without distracting too much from the primary line of discourse. In those circumstances I will offset the comments and precede them with the word “Details,” such as is done with this paragraph, so that you know the nature of what is to follow. These details will sometimes concern the *syntax* of Haskell (i.e., the *notation* used to write Haskell programs) or its *semantics* (i.e., how to calculate with the language features).

1.2 Expressions, Values, and Types

In this section we will take a much closer look at the idea of computation by calculation. In Haskell, the objects that we perform calculations on are called *expressions*, and the objects that result from a calculation (i.e., “the answers”) are called *values*. It is helpful to think of a value just as an expression on which no more calculation can be carried out.

Examples of expressions include *atomic* (meaning indivisible) expressions such as the integer 42 and the character ‘a,’ as well as *structured* (meaning made from smaller pieces) expressions such as the list [1, 2, 3] and the pair (‘b,’ 4) (lists and pairs are different in a subtle way, to be described later). Each of these examples is also a value, because by themselves there is no calculation that can be carried out. As another example,

$1 + 2$ is an expression, and one step of calculation yields the expression 3 , which is a value, because no more calculations can be performed on it.

Sometimes, however, an expression has only a never-ending sequence of calculations. For example, if x is defined as:

$$x = x + 1$$

then here's what happens when we try to calculate the value of x :

$$\begin{aligned} x & \\ \Rightarrow & x + 1 \\ \Rightarrow & (x + 1) + 1 \\ \Rightarrow & ((x + 1) + 1) + 1 \\ \Rightarrow & (((x + 1) + 1) + 1) + 1 \\ & \dots \end{aligned}$$

This is clearly a never-ending sequence of steps, in which case we say that the expression does not terminate, or is *nonterminating*. In such cases, the symbol \perp , pronounced “bottom,” is used to denote the value of the expression.

Every expression (and therefore every value) also has an associated *type*. You can think of types as sets of expressions (or values) in which members of the same set have much in common. Examples include the atomic types *Integer* (the set of all fixed-precision integers) and *Char* (the set of all characters), as well as the structured types $[Integer]$ (the set of all lists of integers) and $(Char, Integer)$ (the set of all character/integer pairs). The association of an expression or value with its type is very important, and there is a special way of expressing it in Haskell. Using the examples of values and types above, we write:

$$\begin{aligned} 42 & \quad . \quad :: \textit{Integer} \\ 'a' & \quad \quad :: \textit{Char} \\ [1, 2, 3] & \quad :: [\textit{Integer}] \\ ('b', 4) & \quad :: (\textit{Char}, \textit{Integer}) \end{aligned}$$

DETAILS

Literal characters are written enclosed in single forward quotes, as in $'a'$, $'A'$, $'b'$, $'\,'$, $'!'$, $' '$ (a space), etc. (There are some exceptions, however; see the *Haskell Report* for details.)

The “ $::$ ” should be read “has type,” as in “42 has type *Integer*.”

DETAILS

Note that the names of specific types are capitalized, such as *Integer* and *Char*, but the names of values are not, such as *simple* and *x*. This is not just a convention; it is required when programming in Haskell. In addition, the case of the other characters matters. For example, *test*, *teSt*, and *tEST* are all distinct names for values, as are *Test*, *TeST*, and *TEST* for types.

Haskell's *type system* ensures that Haskell programs are *well-typed*; that is, that the programmer has not mismatched types in some way. For example, it does not make much sense to add together two characters, so the expression *'a' + 'b'* is *ill-typed*. The best news is that Haskell's type system will tell you if your program is well-typed *before you run it*. This is a big advantage, because most programming errors are manifested as typing errors.

The idea of dividing the world of values into types should be familiar to most people. We do it all the time for just about every kind of object. Take boxes, for example. Just as we have integers and reals, lists and tuples, etc., we also have large boxes and small boxes, cardboard boxes and wooden boxes, and so on. And just as we have lists of integers and lists of characters, we also have boxes of nails and boxes of shoes. And just as we would not expect to be able to take the square of a list or add two characters, we would not expect to be able to use a box to pay for our groceries.

Types help us to make sense of the world by organizing it into groups of common shape, size, functionality, and others. The same is true for programming, where types help us to organize values into groups of common shape, size, and functionality, among others. Of course, the kinds of commonality between values will not be the same as those between objects in the real world, and in general, we will be more restricted - and more formal - about just what we can say about types and how we say it.

1.3 Function Types and Type Signatures

What should the type of a function be? It seems that it should at least convey the fact that a function takes values of one type - *T1*, say - as input and returns values of (possibly) some other type - *T2*, say - as output. In Haskell this is written $T1 \rightarrow T2$, and we say that such a function "maps values of type *T1* to values of type *T2*." If there is more than one argument, the notation is extended with more arrows. For example, if our intent is that the function *simple* defined in the previous section has type $Integer \rightarrow Integer \rightarrow Integer \rightarrow Integer$, we can declare this fact by

including a *type signature* with the definition of *simple*:

```
simple      :: Integer → Integer → Integer → Integer
simple x y z = x * (y + z)
```

DETAILS

When you write Haskell programs using a typical text editor, you will not see nice fonts and arrows as in $Integer \rightarrow Integer$. Rather, you will have to type `Integer -> Integer`.

Haskell's type system also ensures that user-supplied type signatures, such as this one, are correct. Actually, Haskell's type system is powerful enough to allow us to avoid writing any type signatures at all, in which case we say that the type system *infers* the correct types for us.⁵ Nevertheless, judicious placement of type signatures, as we did for *simple*, is a good habit, because type signatures are an effective form of documentation and help bring programming errors to light. Also, in almost every example in this text, I will make a habit of first talking about the types of expressions and functions as a way to better understand the problem at hand, organize our thoughts, and lay down the first ideas of a solution.

The normal use of a function is referred to as *function application*. For example, *simple* 3 9 5 is the application of the function *simple* to the arguments 3, 9, and 5.

DETAILS

Some functions, such as (+), are applied using what is known as *infix syntax*; that is, the function is written between the two arguments rather than in front of them (compare $x + y$ to $f x y$). Infix functions are often called *operators* and are distinguished by the fact that they do not contain any numbers or letters of the alphabet. Thus $\$^!$ and $*\#:$ are infix operators, whereas *thisIsAFunction* and $f9g$ are not (but are still valid names for functions or other values). The only exception to this is that the symbol ' is considered to be alphanumeric; thus f' and *one's* are valid names, but not operators.

In Haskell, when referring to an operator as a value, it is enclosed in parentheses, such as when declaring its type, as in:

```
(+) :: Integer → Integer → Integer
```

⁵ There are a few exceptions to this rule, and in the case of *simple* the inferred type is actually a bit more general than that written above. Both of these points will be returned to later.

Also, when trying to understand an expression such as $f\ x + g\ y$, there is a simple rule to remember: Function application always has “higher precedence” than operator application, so that $f\ x + g\ y$ is the same as $(f\ x) + (g\ y)$.

Despite all of these syntactic differences, however, operators are still just functions.

Exercise 1.3 Identify the well-typed expressions in the following and, for each, give its proper type:

$[(2, 3), (4, 5)]$
 $['z', 42]$
 $('z', -42)$
simple 'a' 'b' 'c'
(simple 1 2 3, simple)

1.4 Abstraction, Abstraction, Abstraction

The title of this section answers the question: “What are the three most important ideas in programming?” Well, perhaps this is an overstatement, but I hope that I’ve gotten your attention, at least. Webster defines the verb “abstract” as follows:

abstract, *vt* (1) remove, separate (2) to consider apart from application to a particular instance.

In programming we do this when we see a repeating pattern of some sort and wish to “separate” that pattern from the “particular instances” in which it appears. Let’s refer to this process as the *abstraction principle* and see how it might manifest itself in problem solving.

1.4.1 Naming

One of the most basic ideas in programming – for that matter, in everyday life – is to *name* things. For example, because it is inconvenient to retype (or remember) the value of π beyond a small number of digits, we may wish to give it a name. In mathematics the Greek letter π in fact *is* the name for this value, but unfortunately we don’t have the luxury of using Greek letters on standard computer keyboards and text editors. So in Haskell we write:

```

pi :: Float
pi = 3.14159

```

to associate the name *pi* with the number 3.14159. The second line above is called an *equation*. The type signature in the first line declares *pi* to be a *floating-point number*, which mathematically, and in Haskell, is distinct from an integer.⁶ Now we can use the name *pi* in expressions whenever we want; it is an abstract representation, if you will, of the number 3.14159. Furthermore, if we ever need to change a named value (which hopefully won't ever happen for *pi*, but could certainly happen for other values), we would only have to change it in one place, instead of in the possibly large number of places where it is used.

Suppose now that we are working on a problem whose solution requires writing some expression more than once. For example, we might find ourselves computing something such as:

$$\begin{aligned} x &:: \text{Float} \\ x &= f (a - b + 2) + g y (a - b + 2) \end{aligned}$$

The first line declares *x* to be a floating-point number, while the second is an equation that defines the value of *x*. Note on the right-hand side of this equation that the expression $a - b + 2$ is repeated – it has two instances – and thus, applying the abstraction principle, we wish to separate it from these instances. We already know how to do this – it's called *naming* – so we might choose to rewrite the single equation above as two:

$$\begin{aligned} c &= a - b + 2 \\ x &= f c + g y c \end{aligned}$$

If, however, the definition of *c* is not intended for use elsewhere in the program, then it is advantageous to “hide” the definition of *c* within the definition of *x*. This will avoid cluttering up the namespace, and prevents *c* from clashing with some other value named *c*. To achieve this, we simply use a **let** expression:

$$\begin{aligned} x &= \mathbf{let} \ c = a - b + 2 \\ &\quad \mathbf{in} \ f c + g y c \end{aligned}$$

A **let** expression restricts the *visibility* of the names that it creates to the internal workings of the **let** expression itself. For example, if we write:

$$\begin{aligned} c &= 42 \\ x &= \mathbf{let} \ c = a - b + 2 \\ &\quad \mathbf{in} \ f c + g y c \end{aligned}$$

then there is no conflict of names; the “outer” *c* is completely different

⁶ I will have more to say about floating-point numbers later in this chapter.

from the “inner” one enclosed in the **let** expression. Think of the inner *c* as analogous to the first name of someone in your household. If your brother’s name is “John” he will not be confused with John Thompson who lives down the street when you say, “John spilled the milk.”

DETAILS

An equation such as $c = 42$ is called a *binding*. A simple rule to remember when programming in Haskell is never to give more than one binding for the same name in a context where the names can be confused, whether at the top level of your program or nested within a **let** expression. For example, this is not allowed:

```
a = 42
a = 43
```

nor is this:

```
a = 42
b = 43
a = 44
```

So you can see that naming - using either top-level equations or equations within a **let** expression - is an example of the abstraction principle in action. It’s often the case, of course, that we *anticipate* the need for abstraction; for example, directly writing down the final solution above, because we knew that we would need to use the expression $a - b + 2$ more than once.

1.4.2 Functional Abstraction

Let’s now consider a more complex example. Suppose we are computing the sum of the areas of three circles with radii r_1 , r_2 , and r_3 , as expressed by

```
totalArea :: Float
totalArea = pi * r1^2 + pi * r2^2 + pi * r3^2
```

DETAILS

([^]) is Haskell’s integer exponentiation operator. In mathematics we would write $\pi \times r^2$ or just πr^2 instead of $pi * r^2$.

Although there isn't an obvious repeating expression here as there was in the last example, there is a repeating *pattern of operations*, namely, the operations that square some given quantity – in this case the radius – and then multiply the result by π . To abstract a sequence of operations such as this, we use a *function*, which we will give the name *circleArea*, that takes the “given quantity” – the radius – as an argument. There are three instances of the pattern, each of which we can expect to replace with a call to *circleArea*. This leads to:

$$\begin{aligned} \text{circleArea} &:: \text{Float} \rightarrow \text{Float} \\ \text{circleArea } r &= \pi * r^2 \\ \text{totalArea} &= \text{circleArea } r1 + \text{circleArea } r2 + \text{circleArea } r3 \end{aligned}$$

Using the idea of **unfolding** described earlier, it is easy to verify that this definition is equivalent to the previous one.

This application of the abstraction principle is sometimes called *functional abstraction*, because the sequence of operations is abstracted as a function, in this case *circleArea*. Actually, it can be seen as a generalization of the previous kind of abstraction: *naming*. That is, *circleArea r1* is just a name for $\pi * r1^2$, *circleArea r2* for $\pi * r2^2$, and *circleArea r3* for $\pi * r3^2$. In other words, a named quantity, such as *c* or *pi* defined previously, can be thought of as a function with no arguments.

Note that *circleArea* takes a radius (a floating-point number) as an argument and returns the area (also a floating-point number) as a result. This is reflected in its type signature.

The definition of *circleArea* could also be hidden within *totalArea* using a **let** expression as we did in the previous example:

$$\begin{aligned} \text{totalArea} &= \text{let } \text{circleArea } r = \pi * r^2 \\ &\quad \text{in } \text{circleArea } r1 + \text{circleArea } r2 + \text{circleArea } r3 \end{aligned}$$

On the other hand, it is more likely that computing the area of a circle will be useful elsewhere in the program, so leaving the definition at the top level is probably preferable in this case.

1.4.3 Data Abstraction

The value of *totalArea* is the sum of the areas of three circles. But what if in another situation we must add the areas of five circles, or in other situations, even more? In situations where the number of things is not certain, it is useful to represent them in a *list* whose length is arbitrary.

So imagine that we are given an entire list of circle areas whose length isn't known when we write the program. What now?

I will define a function *listSum* to add the elements of a list. Before doing so, however, there is a bit more to say about lists.

Lists are an example of a *data structure*, and when their use is motivated by the abstraction principle, I will say that we are applying *data abstraction*. Earlier we saw the example `[1, 2, 3]` as a list of integers, whose type is thus `[Integer]`. Not surprisingly, a list with *no* elements is written `[]`, and pronounced “nil.” To add a single element *x* to the front of a list *xs*, we write `x : xs`. (Note the naming convention used here; *xs* is the plural of *x*, and should be read that way.) In fact, the list `[1, 2, 3]` is equivalent to `1 : (2 : (3 : []))`, which can also be written `1 : 2 : 3 : []` because the infix operator `(:)` is “right associative.”

DETAILS

In mathematics we rarely worry about whether the notation $a + b + c$ stands for $(a + b) + c$ (in which case $+$ would be “left associative”) or $a + (b + c)$ (in which case $+$ would “right associative”). This is because in situations where the parentheses are left out the operator usually is *mathematically* associative, meaning that it doesn't matter which interpretation we choose. If the interpretation *does* matter, mathematicians will include parentheses to make it clear. Furthermore, in mathematics there is an implicit assumption that some operators have higher *precedence* than others; for example, $2 \times a + b$ is interpreted as $(2 \times a) + b$, not $2 \times (a + b)$.

In most programming languages, including Haskell, each operator is defined as having some precedence level and to be either left or right associative. For arithmetic operators, mathematical convention is usually followed; for $2 * a + b$ is interpreted as $(2 * a) + b$ in Haskell. The predefined list-forming operator `(:)` is defined to be right associative. Just as in mathematics, this associativity can be overridden by using parentheses: thus $(a : b) : c$ is a valid Haskell expression (assuming that it is well-typed), and is very different from $a : b : c$. I will explain later how to specify the associativity and precedence of new operators that we define.

Examples of predefined functions defined on lists in Haskell include *head* and *tail*, which return the “head” and “tail” of a list, respectively. That is, $head(x : xs) \Rightarrow x$ and $tail(x : xs) \Rightarrow xs$ (we will define these two functions formally in Section 5.1). Another example is the function `(++)`, which *concatenates*, or *appends*, together its two list arguments. For example, $[1, 2, 3] ++ [4, 5, 6] \Rightarrow [1, 2, 3, 4, 5, 6]$ (`(++)` will be defined in Section 11.2).

Returning to the problem of defining a function to add the elements of a list, let's first express what its type should be:

$$\text{listSum} :: [\text{Float}] \rightarrow \text{Float}$$

Now we must define its behavior appropriately. Often in solving problems such as this, it is helpful to consider, one by one, all possible cases that could arise. To compute the sum of the elements of a list, what might the list look like? The list could be empty, in which case the sum is surely 0. So we write:

$$\text{listSum} [] = 0$$

The other possibility is that the list *isn't* empty (i.e., it contains at least one element) in which case the sum is the first number plus the sum of the remainder of the list. So we write:

$$\text{listSum} (x : xs) = x + \text{listSum} xs$$

Combining these two equations with the type signature brings us to the complete definition of the function *listSum*:

$$\begin{aligned} \text{listSum} &:: [\text{Float}] \rightarrow \text{Float} \\ \text{listSum} [] &= 0 \\ \text{listSum} (x : xs) &= x + \text{listSum} xs \end{aligned}$$

DETAILS

Although intuitive, this example highlights an important aspect of Haskell: *pattern matching*. The left-hand sides of the equations contain *patterns* such as `[]` and `x : xs`. When a function is applied, these patterns are *matched* against the argument values in a fairly intuitive way (`[]` only matches the empty list, and `x : xs` will successfully match any list with at least one element, while naming the first element `x` and the rest of the list `xs`). If the match succeeds, the right-hand side is evaluated and returned as the result of the application. If it fails, the next equation is tried, and if all equations fail, an error results. All of the equations that define a particular function must appear together, one after the other.

Defining functions by pattern matching is quite common in Haskell, and you should eventually become familiar with the various kinds of patterns that are allowed; see Appendix B for a concise summary.

This is called a *recursive function definition* because *listSum* “refers to itself” on the right-hand side of the second equation. Recursion is a very

powerful technique that will be used many times in this text. It is also an example of a general problem-solving technique where a large problem is broken down into many simpler but similar problems; solving these simpler problems one by one leads to a solution to the larger problem.

Here is an example of *listSum* in action:

```
listSum [1, 2, 3]
⇒ listSum (1 : (2 : (3 : [ ])))
⇒ 1 + listSum (2 : (3 : [ ]))
⇒ 1 + (2 + listSum (3 : [ ]))
⇒ 1 + (2 + (3 + listSum [ ]))
⇒ 1 + (2 + (3 + 0))
⇒ 1 + (2 + 3)
⇒ 1 + 5
⇒ 6
```

The first step above is not really a calculation, but rather is a rewriting of the list syntax. The remaining calculations consist of four unfold steps followed by three integer additions.

Given this definition of *listSum* we can rewrite the definition of *totalArea* as:

```
totalArea = listSum [circleArea r1, circleArea r2, circleArea r3]
```

This may not seem like much of an improvement, but if we were adding many such circle areas in some other context, it would be. Indeed, lists are arguably the most commonly used structured data type in Haskell. In the next chapter we will see a more convincing example of the use of lists; namely, to represent the vertices that make up a polygon. Because a polygon can have an arbitrary number of vertices, using a data structure such as a list seems like just the right approach.

In any case, how do we know that this version of *totalArea* behaves the same as the original one? By calculation, of course:

```
listSum [circleArea r1, circleArea r2, circleArea r3]
⇒ { unfold listSum (four successive times) }
circleArea r1 + circleArea r2 + circleArea r3 + 0
⇒ { unfold circleArea (three places) }
pi * r1^2 + pi * r2^2 + pi * r3^2 + 0
⇒ { simple arithmetic }
pi * r1^2 + pi * r2^2 + pi * r3^2
```

1.5 Code Reuse and Modularity

There doesn't seem to be much repetition in our last definition for *totalArea*, so perhaps we're done. In fact, let's pause for a moment and consider how much progress we've made. We started with the definition:

```
totalArea = pi * r12 + pi * r22 + pi * r32
```

and ended with:

```
totalArea = listSum [circleArea r1, circleArea r2, circleArea r3]
```

But we have also introduced definitions for the auxiliary functions *circleArea* and *listSum*. In terms of size, our final program is actually larger than what we began with! So have we actually improved things?

From the standpoint of “removing repeating patterns,” we certainly have, and we could argue that the resulting program is easier to understand. But there is more. Now that we have defined auxiliary functions, such as *circleArea* and *listSum*, we can *reuse* them in other contexts. Being able to reuse code is also called *modularity*, because the reused components are like little modules, or bricks, that can form the foundation of many applications.⁷ We've already talked about reusing *circleArea*; and *listSum* is surely reusable: imagine a list of grocery item prices, or class sizes, or city populations, for each of which we must compute the total. In later chapters you will learn other concepts – most notably higher-order functions and polymorphism – that will substantially increase your ability to reuse code.

1.6 Beware of Programming with Numbers

In mathematics there are many different kinds of number systems. For example, there are integers, natural numbers (i.e., non-negative integers), real numbers, rational numbers, and complex numbers. These number systems possess many useful properties, such as the fact that multiplication and addition are commutative, and that multiplication distributes over addition. You have undoubtedly learned many of these properties in your studies and have used them often in algebra, geometry, trigonometry, and physics, among others.

Unfortunately, each of these number systems places great demands on computer systems. In particular, a number can in general require an

⁷ “Code reuse” and “modularity” are important software engineering principles.

arbitrary amount of memory to represent it – even an infinite amount! Clearly, for example, we cannot represent an irrational number such as π exactly; the best we can do is approximate it, or possibly write a program that computes it to whatever (finite) precision we need in a given application. But even integers (and therefore rational numbers) present problems, because any given integer can be arbitrarily large.

Most programming languages do not deal with these problems very well. In fact, most programming languages do not have exact forms of any of these number systems. Haskell does slightly better than most, in that it has exact forms of integers (the type *Integer*) as well as rational numbers (the type *Rational*, defined in the Ratio Library). But in Haskell and most other languages, there is no exact form of real numbers, for example, which are instead approximated by *floating-point numbers* with either single-word precision (*Float* in Haskell) or double-word precision (*Double*). What’s worse, the behavior of arithmetic operations on floating-point numbers can vary somewhat depending on the CPU being used, although hardware standardization in recent years has lessened the degree of this problem.

The bottom line is that, as simple as numbers seem, great care must be taken when programming with them. Many computer errors, some quite serious and renowned, have been rooted in numerical incongruities. The field of mathematics known as *numerical analysis* is concerned precisely with these problems, and programming with floating-point numbers in sophisticated applications often requires a good understanding of numerical analysis to devise proper algorithms and write correct programs.

As a simple example of this problem, consider the distributive law, expressed here as a calculation in Haskell and used earlier in this chapter in calculations involving the function *simple*:

$$a * (b + c) \Rightarrow a * b + a * c$$

For most floating-point numbers, this law is perfectly valid. For example, in the Hugs implementation of Haskell, the expressions $pi * (3.52 + 4.75)$ and $pi * 3.52 + pi * 4.75$ both yield the same result: 25.981. But funny things can happen when the magnitude of $b + c$ differs significantly from the magnitude of either b or c . For example, the following two calculations are from Hugs:

$$\begin{aligned} 5 * (-0.123456 + 0.123457) &\Rightarrow 4.99189e - 006 \\ 5 * (-0.123456) + 5 * (0.123457) &\Rightarrow 5.00679e - 006 \end{aligned}$$

Although the error here is small, its very existence is worrisome, and in certain situations it could be disastrous. The nature of floating-point

numbers will not be discussed much further in this text, but just remember that they are *approximations* to the real numbers. If real-number accuracy is important to your application, further study of the nature of floating-point numbers is probably warranted.

On the other hand, the distributive law (and many others) is valid in Haskell for the exact data types *Integer* and *Ratio Integer* (i.e., rational numbers). However, another problem arises: Although the representation of an *Integer* in Haskell is not normally something that we are concerned about, it should be clear that the representation must be allowed to grow to an arbitrary size. For example, Haskell has no problem with the following number:

```
veryBigNumber :: Integer
veryBigNumber = 43208345720348593219876512372134059
```

and such numbers can be added, multiplied, etc., without any loss of accuracy. However, such numbers cannot fit into a single word of computer memory, most of which are limited to 32 bits. Worse, because the computer system does not know ahead of time exactly how many words will be required, it must devise a dynamic scheme to allow just the right number of words to be used in each case. The overhead of implementing this idea unfortunately causes programs to run slower.

For this reason, Haskell provides another integer data type called *Int*, which has maximum and minimum values that depend on the word size of the CPU. In other words, every value of type *Int* fits into one word of memory, and the primitive machine instructions for integers can be used to manipulate them very efficiently.⁸ Unfortunately, this means that *overflow* or *underflow* errors could occur when an *Int* value exceeds either the maximum or minimum values. However, most implementations of Haskell (as well as most other languages) do not even tell you when this happens. For example, in Hugs, the following *Int* value:

```
i :: Int
i = 1234567890
```

works just fine, but if you multiply it by 2, Hugs returns the value -1825831516 ! This is because twice *i* exceeds the maximum allowed

⁸ The *Haskell Report* requires that every implementation support *Ints* in the range -2^{29} to $2^{29} - 1$, inclusive. The Hugs implementation running on a Pentium processor, for example, supports the range -2^{31} to $2^{31} - 1$.