The Haskell School of Expression

LEARNING FUNCTIONAL PROGRAMMING THROUGH MULTIMEDIA

PAUL HUDAK

Copyrighted material

The Haskell School of Expression

LEARNING FUNCTIONAL PROGRAMMING THROUGH MULTIMEDIA

PAUL HUDAK
Yale University

Copyrighted material

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

© 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* LaTeX 2ε [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

Copyrighted material

Contents

Preface — *page* xiii

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

Copyrighted material

5.2 Abstraction Over Recursive Definitions 58
5.2.1 Map is Polymorphic 59
5.2.2 Using *map* 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

Copyrighted material

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

Copyrighted material

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 *switch* 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

Copyrighted material

19 An Imperative Robot Language 265
19.1 IRL by Example 266
19.2 *Robot* 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 *Performance* 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

Copyrighted material

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

Bibliography 353

Index 357

Copyrighted material

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

Copyrighted material

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

Copyrighted material

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

Copyrighted material

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

Copyrighted material

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

Copyrighted material

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:

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

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:

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

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:

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

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 $e \Longrightarrow v$ (this arrow is longer than that used earlier). So we can say directly, for example, that $\mathit{simple}\ 3\ 9\ 5 \Longrightarrow 42$, 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, $\mathit{simple}\ 4\ 3\ 2 \Longrightarrow 20$.

We can also use calculation to *prove properties* about programs. For example, it should be clear that for any a, b, and c, $\mathit{simple}\ a\ b\ c$ should yield the same result as $\mathit{simple}\ a\ c\ b$. 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:

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

Copyrighted material

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

Copyrighted material

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 *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,

Copyrighted material

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:

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

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 $\bot$, 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:

```
42        :: Integer
'a'       :: Char
[1, 2, 3] :: [Integer]
('b', 4)  :: (Char, Integer)
```

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*."

Copyrighted material

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 $T1 \to T2$, 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 $Integer \to Integer \to Integer \to Integer$, we can declare this fact by

Copyrighted material

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:

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

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:

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

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:

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

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.

Copyrighted material

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:

$$
\begin{aligned}
a &= 42 \\
a &= 43
\end{aligned}
$$

nor is this:

$$
\begin{aligned}
a &= 42 \\
b &= 43 \\
a &= 44
\end{aligned}
$$

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 $r1$, $r2$, and $r3$, as expressed by

$$
\begin{aligned}
\mathit{totalArea} &:: \mathit{Float} \\
\mathit{totalArea} &= \mathit{pi} * r1\hat{\ }2 + \mathit{pi} * r2\hat{\ }2 + \mathit{pi} * r3\hat{\ }2
\end{aligned}
$$

DETAILS

($\hat{\ }$) is Haskell's integer exponentiation operator. In mathematics we would write $\pi \times r^2$ or just πr^2 instead of $\mathit{pi} * r \hat{\ } 2$.

Copyrighted material

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:

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

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 $\mathit{pi} * r1\hat{\ }2$, *circleArea* $r2$ for $\mathit{pi} * r2\hat{\ }2$, and *circleArea* $r3$ for $\mathit{pi} * r3\hat{\ }2$. 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:

$$\begin{aligned} \mathit{totalArea} = \ &\mathbf{let}\ \mathit{circleArea}\ r = \mathit{pi} * r\hat{\ }2 \\ &\mathbf{in}\ \mathit{circleArea}\ r1 + \mathit{circleArea}\ r2 + \mathit{circleArea}\ r3 \end{aligned}$$

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.

Copyrighted material

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] ⇒ [1, 2, 3, 4, 5, 6] ((++) will be defined in Section 11.2).

Copyrighted material

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

$$listSum :: [Float] \rightarrow Float$$

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:

$$listSum\ [\,] = 0$$

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:

$$listSum\ (x : xs) = x + listSum\ xs$$

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

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

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

Copyrighted material

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:

$$5 * (-0.123456 + 0.123457) \Rightarrow 4.99189e - 006$$
$$5 * (-0.123456) + 5 * (0.123457) \Rightarrow 5.00679e - 006$$

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

Copyrighted material

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 minumum 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 *Int*s 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 $2^{31}-1$.

Copyrighted material

value, so the resulting bits become nonsensical[9] and are interpreted in this case as a negative number of the given magnitude.

This is alarming! Indeed, why should anyone ever use *Int* when *Integer* is available? The answer, as mentioned earlier, is efficiency, but clearly, care should be taken when making this choice. If you are indexing into a list, for example, and you are confident that you are not performing index calculations that might result in the above kind of error, then *Int* should work just fine, because a list longer than 2^{31} will not fit into memory anyway! But if you are calculating the number of microseconds in some large time interval or counting the number of people living on earth, then *Integer* would most likely be a better choice. Choose your number data types wisely!

In this text the data types *Integer*, *Int*, *Float*, and *Rational* will be used for a variety of different applications; for a discussion of the other number types, consult the *Haskell Report*. As I use these data types, I will do so without much discussion; this is not, after all, a book on numerical analysis. But I will issue a warning whenever reasoning about numbers in a way that might not be technically sound.

[9] Actually, they are perfectly sensible in the following way: The 32-bit binary representation of i is 01001001100101100000001011010010, and twice that is 10010011001011000000010110100100. But the latter number is seen as negative because the 32nd bit (the highest-order bit on the CPU on which this was run) is a one, which means it is a negative number in "twos-complement" representation. The twos-complement of this number is in turn 01101100110100111111101001011100, whose decimal representation is 1825831516.

Copyrighted material

CHAPTER TWO

A Module of Shapes: Part I

In the previous chapter you learned quite a few techniques for problem solving via calculation in Haskell. It's time now to apply these ideas to a larger example, which will require learning even more problem-solving skills and Haskell language features.

Our job will be to design a simple module of geometric *shapes*, that is, a collection of functions and data types for computing with geometric shapes such as circles, squares, triangles, and others. Users of this module will be able to create new instances of geometric shapes and compute their areas. You will learn lots of new things in building this module, including how to design your own data types. Then in Chapter 4 we will extend this functionality with the ability to *draw* geometric shapes, and in Chapter 6 compute their *perimeters.*

In the description above I refer to the end product as a *module*, through which a user has access to certain well-defined functionalities. A module can be seen as a way to conveniently wrap up an application in such a way that only the functionality intended for the end-user is visible; everything else needed to implement the system is effectively hidden.

In Haskell we can create a module named *Shape* in the following way:

module *Shape* (· · ·) **where**

...body-of-module...

The "(· · ·)" after the name *Shape* will ultimately be a list of names of the functions and data types that the end-user is intended to use, and is sometimes called the *interface* to a module. At the end of this chapter we will fill in the details of the interface once we know what they should be. The *...body-of-module...* is of course where we will place all the code developed in this chapter.

Copyrighted material

> **DETAILS**
> Module names must always be capitalized (just like type names).

A user of our shape module can later *import* it into a module that he or she is constructing, by writing:

import *Shape*

Indeed, we will do exactly this in later chapters where new modules will be created in which users will be able to draw shapes, compute their perimeters, combine them into larger "regions," color them, scale them, and place them on a "virtual desktop." This desktop will be displayed on your computer screen, and will be designed in such a way that regions will rise to the surface of the desktop when they are clicked, just like windows do in a windows-based user-interface.

2.1 Geometric Shapes

Our first job will be to design a single data type to represent all of the possible geometric shapes of interest to us. Aside from the fact that this is intuitively appealing, there are several pragmatic reasons for doing so. For example, in Section 1.4.2 we saw a function for computing the area of a circle, and later a function for computing the area of a square. If we were to define functions for computing the areas of, say, *n* different shapes, we would end up with *n* functions, each with a different name. Similarly, we would have *n* functions for drawing shapes and *n* functions for computing their perimeters. In contrast, if we had a single data type that captured *all* of the geometric shapes, we could (hopefully) define a single function for each of these tasks.

In Haskell, new data types such as this are defined using a **data** declaration:

```
data Shape = Circle Float
           | Square Float
```

This declaration can be read: "There are two kinds of *Shapes*: a *circle* of the form *Circle r* where *r* is a radius of type *Float*, and a *square* of the form *Square s* where *s* is the length (also of type *Float*) of one side." Because *Circle* and *Square* construct new values in this data type, they are called *constructors*.

Copyrighted material

But, you might point out, with the *Polygon* constructor we can create polygons with an arbitrary number of sides, each with an arbitrary length, so why include the special cases of rectangle and right triangle? In other words, why not just define functions *rectangle* and *rtTriangle* in terms of *Polygon*? Good question! One answer, as mentioned earlier, is in the interest of pedagogy: I am trying to illustrate a variety of programming techniques. Another answer is that the algorithms for computing the areas of rectangles and right triangles are simpler than the algorithm for computing the area of a general polygon. A final answer is that there may be lower-level graphics commands, say, that are more efficient at drawing rectangles and/or right triangles than general polygons. In any case, let's proceed with our design and investigate its consequences later.

Exercise 2.1 Define functions *rectangle* and *rtTriangle* in terms of *Polygon*.

Exercise 2.2 Define a function *regularPolygon* :: *Int* → *Side* → *Shape* such that *regularPolygon n s* is a regular polygon with *n* sides, each of length *s*. (Hint: Consider using some of Haskell's trigonometric functions, such as *sin* :: *Float* → *Float*, *cos* :: *Float* → *Float*, and *tan* :: *Float* → *Float*.)

2.2 Areas of Shapes

Returning to the problem in Section 1.4.2 of computing areas, we will define a function:

area :: *Shape* → *Float*

by defining its behavior on each of the *Shape* constructors. We know how to do this for a rectangle:

area (*Rectangle* $s1$ $s2$) = $s1 * s2$

And the area of a right triangle is just as easy:

area (*RtTriangle* $s1$ $s2$) = $s1 * s2/2$

Before proceeding, note that this way of writing function definitions - by *pattern matching* on the arguments - is just like the way we defined the function *listSum* in Section 1.4.3. There, *listSum* was defined by its behavior on the two constructors in the list data types, [ ] and (:). In the case of the *Shape* data type, we happen to have four constructors to consider instead of two.

Copyrighted material

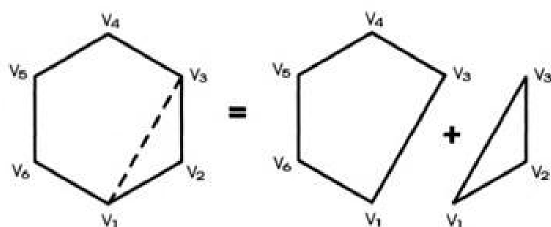

Figure 2.1. Computing the Area of a Convex Polygon

Moving on, a standard geometry text tells us that the area of an ellipse with radii r_1 and r_2 is just $\pi r_1 r_2$. It is easy to see that this reduces to πr^2 for a circle. Translated into Haskell, this becomes:

```
area (Ellipse r1 r2) = pi * r1 * r2
```

What about the area of a general polygon? If the polygon is *convex* (meaning that all of its interior angles are less than 180°), its area can be computed fairly simply as follows (the case of concave polygons will be an exercise):

1. Compute the area of the triangle formed by the first three vertices of the polygon.
2. Delete the second vertex, forming a new polygon.
3. If there are at least three vertices in the new polygon, repeat this procedure, returning as the total area the sum of the areas of the individual triangles.

The intuition for this algorithm is shown pictorially in Fig. 2.1; it is another example of solving a problem by solving a smaller problem first, which at least slightly reduces the size of the larger one.

Although we haven't yet decided how to compute the area of a general triangle, we can write out the above algorithm in Haskell as follows:

```
area (Polygon (v1 : v2 : v3 : vs))
    = triArea v1 v2 v3 + area (Polygon (v1 : v3 : vs))
area (Polygon _)
    = 0
```

DETAILS

The first line above is an example of a *nested pattern*; that is, the pattern (v1 : v2 : v3 : vs) is nested within the pattern *Polygon* ... .

Copyrighted material

The second equation uses an "underscore" character "_" as a pattern. This is called a *wildcard pattern*, and matches *any* argument. When more than one equation is used to define a function, Haskell will try them in the order that they appear. So in the case of *area*, an attempt is first made to match a list containing at least three elements. If that fails, then the second equation is tried, which of course succeeds immediately, yielding the value 0.

This version of the algorithm has a more "declarative" feel than the description given earlier and can be read: "The area of a (convex) polygon is the area of the triangle formed by its first three vertices, plus the area of the polygon resulting from deleting the second vertex." In general, we will try to write declarative definitions such as this directly.

However, note in the recursive call to *area* that the polygon is "reconstructed" using the *Polygon* constructor; so the polygon is taken apart and then put back together, so to speak, on each recursive call. Also note that the first vertex never disappears; it is always part of the reconstruction. We can avoid these slight inefficiencies by defining an auxiliary function *polyArea* that computes the area directly from a list of vertices, as follows:

```
area (Polygon (v1 : vs)) = polyArea vs
  where polyArea                   :: [Vertex] -> Float
        polyArea (v2 : v3 : vs')    = triArea v1 v2 v3
                                        + polyArea (v3 : vs')
        polyArea _                  = 0
```

DETAILS

Note the use of a **where** expression. For the most part you can consider *exp* **where** ...*equations*... to be equivalent to **let** ...*equations*... **in** *exp*. The only difference is that a **where** expression is only allowed at the top level of a function definition, where its visibility rules are slightly different (see the *Haskell Report* for details).

What about *triArea*? Fortunately, there is a single formula for the area of an arbitrary triangle with sides of lengths a, b, and c, due to Heron:

$$A = \sqrt{s(s-a)(s-b)(s-c)} \quad where\ s = \tfrac{1}{2}(a+b+c)$$

Copyrighted material

or, in Haskell:

```
triArea          :: Vertex → Vertex → Vertex → Float
triArea v1 v2 v3 = let a = distBetween v1 v2
                       b = distBetween v2 v3
                       c = distBetween v3 v1
                       s = 0.5 * (a + b + c)
                   in sqrt (s * (s − a) * (s − b) * (s − c))
```

DETAILS

sqrt is Haskell's square root function.

Also, note that more than one equation is allowed in a **let** (or **where**) expression. The first characters of each equation, however, must line up vertically, and if an equation takes more than one line then the subsequent lines must be to the right of the first characters. For example, this is legal:

```
let a = aLongName
        +anEvenLongerName
    b = 56
in ...
```

but neither of these is:

```
let a = aLongName
  +anEvenLongerName
    b = 56
in ...

let a = aLongName
        +anEvenLongerName
  b = 56
in ...
```

(The second line of the first example is too far to the left, as is the third line in the second example.)

Although this rule, called the *layout rule*, may seem a bit ad hoc, it avoids the use of special syntax to denote the end of one equation and the beginning of the next, thus enhancing readability. In practice, use of layout is rather intuitive. Just remember two things:

1. The first character following either **where** or **let** (and a few other keywords that we will see later) is what determines the starting column for the set

Copyrighted material

of equations. Thus, we can begin the equations on the same line as the keyword, the next line, or whatever.

2. Be sure that the starting column is further to the right than the starting column associated with any immediately surrounding clause (otherwise it would be ambiguous). The "termination" of an equation happens when something appears at or to the left of the starting column associated with that equation.

Here *distBetween* computes the difference between two vertices - that is, the length of each side of the triangle - and is defined by:

```
distBetween :: Vertex → Vertex → Float
distBetween (x1, y1) (x2, y2)
            = sqrt ((x1 − x2)^2 + (y1 − y2)^2)
```

This definition is easy to see as an application of Pythagorean's theorem, as shown graphically in Fig. 2.2.

In summary, by collecting the four equations for *area*, we see that we have covered each of the constructors in the *Shape* data type:

```
area                        :: Shape → Float
area (Rectangle s1 s2)      =  s1 * s2
area (RtTriangle s1 s2)     =  s1 * s2/2
area (Ellipse r1 r2)        =  pi * r1 * r2
area (Polygon (v1 : vs))    =  polyArea vs
  where polyArea                  :: [Vertex] → Float
        polyArea (v2 : v3 : vs')  =  triArea v1 v2 v3
                                       +polyArea (v3 : vs')
        polyArea _                =  0
```

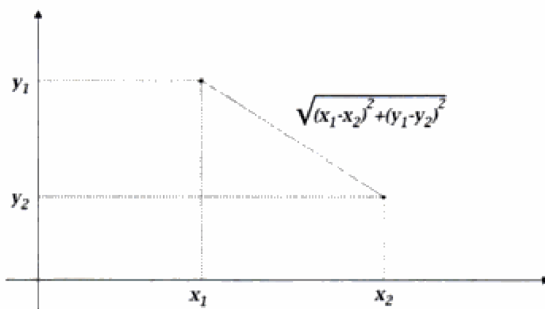

Figure 2.2. Computing the Distance between Two Vertices

Copyrighted material

Simplifying a to s_1 and c to s_2 yields:

$$\sqrt{s(s-s_1)(s-b)(s-s_2)} \tag{2.1}$$

where $b = \sqrt{s_1^2 + s_2^2}$ and $s = \frac{1}{2}(s_1 + b + s_2)$.

We can simplify this further in two steps. First note:

$$\begin{aligned}
& s(s-b) = \tfrac{1}{2}(s_1 + b + s_2)(\tfrac{1}{2}(s_1 - b + s_2)) \\
& = \tfrac{1}{4}(s_1 + s_2 + b)(s_1 + s_2 - b) = \tfrac{1}{4}((s_1 + s_2)^2 - b^2) \\
& = \tfrac{1}{4}((s_1 + s_2)^2 - (s_1^2 + s_2^2)) = \tfrac{1}{4}(s_1^2 + 2s_1s_2 + s_2^2 - (s_1^2 + s_2^2)) \\
& = \tfrac{1}{4}(2s_1s_2) = \tfrac{1}{2}s_1s_2
\end{aligned}$$

And further note:

$$\begin{aligned}
& (s-s_1)(s-s_2) = \tfrac{1}{2}(b - s_1 + s_2)(\tfrac{1}{2}(b + s_1 - s_2)) \\
& = \tfrac{1}{4}(b^2 - (s_1 - s_2)^2) = \tfrac{1}{4}((s_1^2 + s_2^2) - (s_1 - s_2)^2) \\
& = \tfrac{1}{4}((s_1^2 + s_2^2) - (s_1^2 - 2s_1s_2 + s_2^2)) \\
& = \tfrac{1}{2}s_1s_2
\end{aligned}$$

Combining these and continuing from point (1), we get:

$$\begin{aligned}
& \sqrt{s(s-s_1)(s-b)(s-s_2)} \\
& = \sqrt{(\tfrac{1}{2}s_1s_2)(\tfrac{1}{2}s_1s_2)} \\
& = \tfrac{1}{2}s_1s_2
\end{aligned}$$

This last formula is just the area of a right triangle with sides s_1 and s_2. If you compare this mathematical proof with the corresponding proof by calculation in Haskell, you will find that they are almost identical, except for the notation used. So reasoning in Haskell is very often quite the same as reasoning in mathematics. Because we often use mathematics to *specify* the correct behavior of our programs, a proof by calculation in Haskell is a proof both *about the implementation and about the specification.* This is why Haskell is sometimes referred to as an "executable specification language."

Unfortunately, as discussed in Section 1.6, care must be taken when performing this kind of reasoning with floating-point numbers. As is often the case, numbers that do not approach the limits of floating-point precision work perfectly well:

area (*RtTriangle* 3.652 5.126) $\Rightarrow$ 9.36008
area (*Polygon* [(0, 0), (3.652, 0), (0, 5.126)]) $\Rightarrow$ 9.36008

Copyrighted material

whereas ones that do can yield inconsistent results:

area (*RtTriangle* 0.0001 5.126) ⇒ 0.0002563
area (*Polygon* [(0, 0), (0.0001, 0), (0, 5.126)]) ⇒ 0.000256648

Exercise 2.3 Prove the following property:

area (*Rectangle* *s*1 *s*2)
⇒ *area* (*Polygon* [(0, 0), (*s*1, 0), (*s*1, *s*2), (0, *s*2)])

Exercise 2.4 Define a function *convex* :: *Shape* → *Bool* that determines whether or not its argument is a convex shape (although we are mainly interested in the convexity of polygons, you might as well define it for each kind of shape).

Exercise 2.5 Here is an alternative way to compute the area of a polygon. Consider a polygon in quadrant 1 of the Cartesian plane (i.e., every vertex has positive x and y coordinates). Then every pair of adjacent vertices forms a trapeziod with respect to the x-axis. Starting at any vertex and working clockwise, compute these areas one-by-one, counting the area as positive if the x-coordinate increases, and negative if it decreases. The sum of these areas is then the area of the polygon.

It is easy to see that this algorithm is correct for a convex polygon by just looking at an example, as in Fig. 2.3. But the real beauty in this algorithm is that it works for *concave* polygons as well (see figure), and for a polygon located anywhere in the Cartesian plane. It is also more efficient than our previous algorithm. (Why?)

Write a Haskell function to compute polygonal areas in this way.

(Note: Polygons can not only be convex or concave, but also *self-crossing*. Consider, for example, the four-vertex polygon that outlines a "bowtie," or the five-vertex polygon that outlines a five-pointed star. What is the proper notion of area in this case, and do any of the algorithms discussed here compute it properly?)

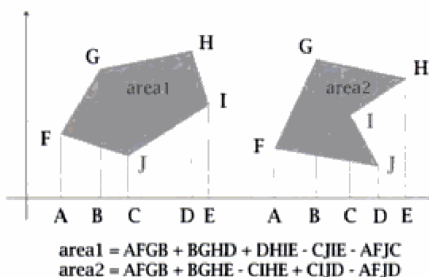

Figure 2.3. Using Trapezoids to Compute Area of Polygon

Copyrighted material

2.3 Cleaning Up

Recall the discussion at the beginning of this chapter about modules and the desire to make visible only those named values that are of interest to the user of a module, thereby hiding irrelevant details of the implementation. We can now fill in the module declaration details to achieve this effect:

```
module Shape (Shape (Rectangle, Ellipse, RtTriangle, Polygon),
              Radius, Side, Vertex,
              square, circle, distBetween, area
             ) where
  ...
```

The list of names above are those entities that are intended to be visible, or *exported*, from the module. These names come in three flavors:

1. Data types (in this case the data type *Shape*) that are listed with their constructors in parentheses.

> **DETAILS**
> A data type and its list of constructors may be abbreviated by just listing the name of the data type followed by (...). For example, the entire *Shape* data type may be exported using the form *Shape* (...). There are many other rules concerning the import and export of entities to and from modules. Some of these rules will be used later in the text, but you should consult the *Haskell Report* for all of the details.

2. Type synonyms (*Radius*, *Side*, and *Vertex*).
3. Ordinary values (in this case the functions *square*, *circle*, *distBetween*, and *area*).

So, for example, the function *triArea* is *not* exported from the *Shape* module, and thus is not available for use by someone importing the module.

> **DETAILS**
> It is sometimes desirable to export *everything* from a module. A shorthand way to express this is to simply write:
>
> ```
> module Shape where
>   ...
> ```
>
> (i.e., omitting the list of exports is equivalent to exporting everything).

Copyrighted material

CHAPTER THREE

Simple Graphics

In this chapter simple *graphics programming* in Haskell will be explained. Graphics in Haskell is consistent with the notion of computation via calculation, although it is special enough to warrant the use of special terminology and notation. In the next chapter we will use the techniques learned here to draw in a graphics window the geometric shapes defined in the last chapter. The ideas developed in this chapter will be put into a module called *SimpleGraphics*:

module *SimpleGraphics* **where**

There are many predefined functions and data types in Haskell, so many, in fact, that they demand some organization. Entities that are deemed essential to defining the fundamental nature of Haskell are contained in what is called the *Standard Prelude*, a collection of modules defining various categories of functionality. Entities that are deemed useful but not essential are contained in one of several *Standard Libraries*, also a collection of modules. The entire Standard Prelude is automatically imported into every program that you write, whereas the Standard Libraries need to be imported module-by-module.

Unfortunately, there is no standard graphics library for Haskell yet, although there is one in popular use on Windows machines called *Graphics*. The basic graphics functionality that we will use is defined in a library called *SOEGraphics*, which is very similar to *Graphics* but is guaranteed to work with this textbook, whereas the *Graphics* library may evolve over time. To use *SOEGraphics*, it must be imported into the module that is using it, as follows:

import *SOEGraphics*

Graphics is a special case of *input/output* (IO) processing in Haskell, and thus I will begin with a discussion of this more general idea.

Copyrighted material

3.1 Basic Input/Output

The Haskell Report defines the result of a program as the value of the name *main* in the module *Main*. On the other hand, the Hugs implementation of Haskell allows you to type whatever expression you wish to the Hugs prompt, and it will evaluate it for you. But in either case, the Haskell system "executes a program" by evaluating an expression, which (for a well-behaved program) eventually yields a value. The system must then display that value on your computer screen in some way that makes sense to you. Most systems will try to display the result in the same way that you would type it in as part of a program. So an integer is printed as an integer, a string as a string, a list as a list, and so on. I will refer to the area of the computer screen where this result is printed as the *standard output area*, which may vary from one implementation to another.

But what if a program is intended to write to a file or print a file on a printer or, the main topic of this chapter, draw a picture in a graphics window? These are examples of *output*, and there are related questions about *input*: For example, how does a program receive input from a keyboard or a mouse?

In general, how does Haskell's "expression-oriented" notion of "computation by calculation" accommodate these various kinds of input and output?

The answer is fairly simple: In Haskell, there is a special kind of value called an *action*. When a Haskell system evaluates an expression that yields an action, it knows not to try to display the result in the standard output area, but rather to "take the appropriate action." There are primitive actions - such as writing a single character to a file or receiving a single character from the keyboard - as well as compound actions - such as printing an entire string to a file. Haskell expressions that evaluate to actions are commonly called *commands*, because they command the Haskell system to perform some kind of action. Haskell functions that yield actions when they are applied are also commonly called commands.

Commands are still just expressions, of course, and some commands return a value for subsequent use by the program: keyboard input, for instance. A command that returns a value of type *T* has type *IO T*; if no useful value is returned the command has type *IO* (). The simplest example of a command is *return x*, which for a value *x* :: *T* immediately returns *x* and has type *IO T*.

DETAILS

The type () is called the *unit type*, and has exactly one value, which is also written (). Thus *return* () has type *IO* (), and is often called a "noop"

Copyrighted material

However, a list of actions is just a list of values; they actually don't *do* anything until they are sequenced appropriately using a **do** expression, and then returned as the value *main* of the overall program. Still, it is often convenient to place actions into a list as above, and the Haskell Report and Libraries have some useful functions for turning them into commands. In particular, the function *sequence_* in the Standard Prelude, when used with IO, has type:

$$sequence_ :: [IO\ a] \rightarrow IO\ ()$$

and can thus be applied to the *actionList* above to yield the single command:

$$main = sequence_\ actionList$$

Before I give you a more interesting example of this idea, I will tell you a secret (more secrets will be revealed later in the text):

DETAILS

Haskell's strings are really *lists of characters.* In other words, *String* is a shorthand – a type synonym – for a list of characters:

$$\textbf{type}\ String = [Char]$$

However, because strings are used so often, Haskell allows you to write "Hello" instead of $['H', 'e', 'l', 'l', 'o']$. But keep in mind that this is just syntax – strings really are just lists of characters, and these two ways of writing this string are identical from Haskell's perspective.

(Earlier the type synonym *FilePath* was defined for *String*. This shows that type synonyms can be made for other type synonyms.)

Now back to the example. From the function $putChar :: Char \rightarrow IO\ ()$, which prints a single character to the standard output area, we can define the function *putStr* used earlier, which prints an entire string. To do this, let's first define a function that converts a list of characters (i.e., a string) into a list of IO actions:

$$\begin{aligned} &putCharList && :: String \rightarrow [IO\ ()] \\ &putCharList\ [\,] && = [\,] \\ &putCharList\ (c : cs) && = putChar\ c : putCharList\ cs \end{aligned}$$

With this, *putStr* is easily defined:

$$\begin{aligned} &putStr && :: String \rightarrow IO\ () \\ &putStr\ s && = sequence_\ (putCharList\ s) \end{aligned}$$

Copyrighted material

Note that the expression *putCharList s* is a list of actions, and *sequence_* is used to turn them into a single (compound) command, just as we did earlier. (The function *putStr* can also be defined directly as a recursive function, but I leave this as an exercise.)

IO processing in Haskell is consistent with everything you have learned about programming with expressions and reasoning through calculation, although that may not be completely obvious yet. Indeed, it turns out that a **do** expression is just syntax for a more primitive way of combining actions using functions. This secret will be revealed in full in Chapter 18.

3.2 Graphics Windows

Let's now look at the particulars of *graphics* IO. Graphics commands are no different in concept from those discussed earlier. However, there is no "standard graphics area" on which to draw things. Instead, we must create a fresh *graphics window*. Furthermore, because we may wish to create several such windows, we need a way to distinguish them once they are created, in order to specify in which window to draw at some particular point in a program. Haskell accomplishes this by returning a unique value of type *Window* at the time we create a window.

To see this concretely, let's look at the type of the *openWindow* command that (you guessed it) opens a window:

```
openWindow  :: Title → Size → IO Window
type Title   = String
type Size    = (Int, Int)
```

The *Title* is a string displayed in the title bar of the new graphics window, and *Size* represents the size of the window as a pair of numbers indicating the width and height in *pixel coordinates*. A pixel is the smallest dot that can be displayed on a computer screen; usually 100 or so pixels can be lined up along one inch. The *Window* that returns from a call to *openWindow* is used in subsequent graphics commands to tell the computer within which window to perform its action. In other words, every call to *openWindow* creates a new, unique window, and the *Window* value provides a way to distinguish between them in the rest of the program.

Let's write our first graphics program:

```
main0
  = runGraphics (
    do w ← openWindow
```

Copyrighted material

```
                "My First Graphics Program" (300, 300)
        drawInWindow w (text (100, 200) "HelloGraphicsWorld")
        k ← getKey w
        closeWindow w
    )
```

It's not hard to guess what this program does: A 300 × 300-pixel graphics window is opened, a greeting message is displayed in it, and the window remains open until the user types a character on the keyboard. The following five new functions are introduced by this example:

- *runGraphics* :: *IO* () → *IO* () runs a graphics "action." This is needed because of special operating system tasks that need to be set up to perform graphics IO.
- *drawInWindow* :: *Window* → *Graphic* → *IO* () draws a given *Graphic* value on a given *Window*.
- *text* :: *Point* → *String* → *Graphic* creates a *Graphic* value consisting of a *String* whose lower left-hand corner is at the location specified by the *Point* argument, in pixel coodinates:

  ```
  type Point = (Int, Int)
  ```

- *getKey* :: *Window* → *IO* () waits for the user to press (and release) a key on the keyboard. In the above example *getKey* is used to prevent the window from closing before the user has a chance to read what's on the screen.
- *closeWindow* :: *Window* → *IO* () closes the specified window.

You should know enough about IO at this point that these descriptions are sufficient to fully understand the sample program given above, except for one other detail: (0, 0) is the location of the *upper left-hand corner of the graphics window.* As the x coordinate increases, the position moves to the right, and as the y coordinate increases, the position moves downward. Thus, in the above program, the bottom right-hand corner of the graphics window is at coordinate (299, 299). I will have more to say about this in the next chapter.

For convenience, I will define the following command, which also demonstrates how to write a *loop* using command sequencing:

```
spaceClose :: Window → IO ()
spaceClose w
        = do k ← getKey w
             if k ==' ' then closeWindow w
                        else spaceClose w
```

Copyrighted material

As a final example of the use of *map*, let's generate a list of eight concentric circles:

$$conCircles \;=\; map\ circle\ [2.4,\ 2.1..0.3]$$

DETAILS
A list $[a, b..c]$ is called an *arithmetic sequence,* and is special syntax for the list $[a, a + d, a + 2 * d, \ldots, c]$ where $d = b - a$. Thus, $[2.4, 2.1..0.3]$ is shorthand for the list $[2.4, 2.1, 1.8, 1.5, 1.2, 0.9, 0.6, 0.3]$.

We can pair each of these circles with a color:

```
coloredCircles =
  zip [Black, Blue, Green, Cyan, Red, Magenta, Yellow, White]
      conCircles
```

DETAILS
The *PreludeList* function *zip* (see Chapter 23) takes two lists and returns a list of the pairwise elements. For example:

$$zip\ [1, 2, 3]\ [4, 5, 6] \;\Rightarrow\; [(1, 4), (2, 5), (3, 6)]$$

It is defined as:

```
zip                 :: (a → b → c) → [a] → [b] → [c]
zip (a : as) (b : bs) = (a, b) : zip as bs
zip _ _             = [ ]
```

(Recall that _ is a wildcard pattern, which matches any argument.)

This list of color/circle pairs can then be drawn using *drawShapes*:

```
main
  = runGraphics (
      do w ← openWindow "Bull's Eye" (xWin, yWin)
         drawShapes w coloredCircles
         spaceClose w
    )
```

Note that *coloredCircles* is ordered largest circle first; thus *drawShapes* draws them in order of decreasing size (otherwise the larger circles would obscure the smaller ones!). A snapshot of *main* is shown in Fig. 5.1.

Copyrighted material

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.

Cover art by Mary Valencia

CAMBRIDGE
UNIVERSITY PRESS
www.cambridge.org

ISBN 0-521-64408-9

Covers Haskell 98

Copyrighted material