

## Programming in Haskell

### Second Edition

Haskell is a purely functional language that allows programmers to rapidly develop clear, concise and correct software. The language has grown in popularity in recent years, both in teaching and in industry. This book is based on the author's experience of teaching Haskell for more than 20 years. All concepts are explained from first principles and no programming experience is required, making this book accessible to a broad spectrum of readers. While Part I focuses on basic concepts, Part II introduces the reader to more advanced topics.

This new edition has been extensively updated and expanded to include recent and more advanced features of Haskell, new examples and exercises, selected solutions, and freely downloadable lecture slides and code. The presentation is clean and simple, while also being fully compliant with the latest version of the language, including recent changes concerning applicative, monadic, foldable and traversable types.

**Graham Hutton** is Professor of Computer Science at the University of Nottingham. He has taught Haskell to thousands of students and received numerous best lecturer awards. Hutton has served as an editor of the *Journal of Functional Programming*, chair of the Haskell Symposium and the International Conference on Functional Programming, vice-chair of the ACM Special Interest Group on Programming Languages, and he is an ACM Distinguished Scientist.

# Programming in Haskell

Second Edition

GRAHAM HUTTON

University of Nottingham

Cambridge University Press  
978-1-316-62622-1 – Programming in Haskell  
Graham Hutton  
Frontmatter  
[More Information](#)

**CAMBRIDGE**  
UNIVERSITY PRESS

University Printing House, Cambridge CB2 8BS, United Kingdom

One Liberty Plaza, 20th Floor, New York, NY 10006, USA  
477 Williamstown Road, Port Melbourne, VIC 3207, Australia  
4843/24, 2nd Floor, Ansari Road, Daryaganj, Delhi - 110002, India  
79 Anson Road, #06-04/06, Singapore 079906

Cambridge University Press is part of the University of Cambridge.

It furthers the University's mission by disseminating knowledge in the pursuit of education, learning, and research at the highest international levels of excellence.

[www.cambridge.org](http://www.cambridge.org)

Information on this title: [www.cambridge.org/9781316626221](http://www.cambridge.org/9781316626221)

10.1017/9781316784099

© Graham Hutton 2007, 2016

This publication 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 2007

Second edition 2016

Printed in the United Kingdom by Clays, St Ives plc in 2016

*A catalogue record for this publication is available from the British Library*

ISBN 978-1-316-62622-1 Paperback

Cambridge University Press has no responsibility for the persistence or accuracy of URLs for external or third-party Internet Web sites referred to in this publication, and does not guarantee that any content on such Web sites is, or will remain, accurate or appropriate.

Cambridge University Press  
978-1-316-62622-1 – Programming in Haskell  
Graham Hutton  
Frontmatter  
[More Information](#)

---

*For Annette, Callum and Tom*

## Contents

	<i>Foreword</i>	<i>page</i> xiii
	<i>Preface</i>	xv
<b>Part I</b>	<b>Basic Concepts</b>	<b>1</b>
<b>1</b>	<b>Introduction</b>	<b>3</b>
	1.1 Functions	3
	1.2 Functional programming	4
	1.3 Features of Haskell	6
	1.4 Historical background	8
	1.5 A taste of Haskell	9
	1.6 Chapter remarks	13
	1.7 Exercises	13
<b>2</b>	<b>First steps</b>	<b>14</b>
	2.1 Glasgow Haskell Compiler	14
	2.2 Installing and starting	14
	2.3 Standard prelude	15
	2.4 Function application	16
	2.5 Haskell scripts	17
	2.6 Chapter remarks	21
	2.7 Exercises	21
<b>3</b>	<b>Types and classes</b>	<b>22</b>
	3.1 Basic concepts	22
	3.2 Basic types	23
	3.3 List types	25
	3.4 Tuple types	26
	3.5 Function types	27
	3.6 Curried functions	27
	3.7 Polymorphic types	29
	3.8 Overloaded types	30
	3.9 Basic classes	31
	3.10 Chapter remarks	36

	3.11 Exercises	36
<b>4</b>	<b>Defining functions</b>	<b>38</b>
	4.1 New from old	38
	4.2 Conditional expressions	38
	4.3 Guarded equations	39
	4.4 Pattern matching	40
	4.5 Lambda expressions	42
	4.6 Operator sections	44
	4.7 Chapter remarks	45
	4.8 Exercises	45
<b>5</b>	<b>List comprehensions</b>	<b>47</b>
	5.1 Basic concepts	47
	5.2 Guards	48
	5.3 The <code>zip</code> function	50
	5.4 String comprehensions	51
	5.5 The Caesar cipher	52
	5.6 Chapter remarks	56
	5.7 Exercises	57
<b>6</b>	<b>Recursive functions</b>	<b>59</b>
	6.1 Basic concepts	59
	6.2 Recursion on lists	61
	6.3 Multiple arguments	63
	6.4 Multiple recursion	64
	6.5 Mutual recursion	65
	6.6 Advice on recursion	66
	6.7 Chapter remarks	71
	6.8 Exercises	71
<b>7</b>	<b>Higher-order functions</b>	<b>73</b>
	7.1 Basic concepts	73
	7.2 Processing lists	74
	7.3 The <code>foldr</code> function	76
	7.4 The <code>foldl</code> function	79
	7.5 The composition operator	81
	7.6 Binary string transmitter	82
	7.7 Voting algorithms	86
	7.8 Chapter remarks	89
	7.9 Exercises	89
<b>8</b>	<b>Declaring types and classes</b>	<b>92</b>
	8.1 Type declarations	92

		8.2 Data declarations	93
		8.3 Newtype declarations	95
		8.4 Recursive types	96
		8.5 Class and instance declarations	99
		8.6 Tautology checker	101
		8.7 Abstract machine	106
		8.8 Chapter remarks	108
		8.9 Exercises	109
<b>9</b>	<b>The countdown problem</b>		<b>111</b>
	9.1 Introduction		111
	9.2 Arithmetic operators		112
	9.3 Numeric expressions		113
	9.4 Combinatorial functions		114
	9.5 Formalising the problem		115
	9.6 Brute force solution		115
	9.7 Performance testing		116
	9.8 Combining generation and evaluation		117
	9.9 Exploiting algebraic properties		118
	9.10 Chapter remarks		119
	9.11 Exercises		120
<b>Part II</b>	<b>Going Further</b>		<b>121</b>
<b>10</b>	<b>Interactive programming</b>		<b>123</b>
	10.1 The problem		123
	10.2 The solution		124
	10.3 Basic actions		125
	10.4 Sequencing		126
	10.5 Derived primitives		127
	10.6 Hangman		128
	10.7 Nim		129
	10.8 Life		133
	10.9 Chapter remarks		137
	10.10 Exercises		137
<b>11</b>	<b>Unbeatable tic-tac-toe</b>		<b>139</b>
	11.1 Introduction		139
	11.2 Basic declarations		140
	11.3 Grid utilities		141
	11.4 Displaying a grid		142
	11.5 Making a move		143
	11.6 Reading a number		144
	11.7 Human vs human		144

x	<b>Contents</b>	
	11.8 Game trees	145
	11.9 Pruning the tree	147
	11.10 Minimax algorithm	148
	11.11 Human vs computer	150
	11.12 Chapter remarks	151
	11.13 Exercises	151
<b>12</b>	<b>Monads and more</b>	<b>153</b>
	12.1 Functors	153
	12.2 Applicatives	157
	12.3 Monads	164
	12.4 Chapter remarks	174
	12.5 Exercises	175
<b>13</b>	<b>Monadic parsing</b>	<b>177</b>
	13.1 What is a parser?	177
	13.2 Parsers as functions	177
	13.3 Basic definitions	179
	13.4 Sequencing parsers	179
	13.5 Making choices	181
	13.6 Derived primitives	183
	13.7 Handling spacing	186
	13.8 Arithmetic expressions	187
	13.9 Calculator	191
	13.10 Chapter remarks	194
	13.11 Exercises	194
<b>14</b>	<b>Foldables and friends</b>	<b>196</b>
	14.1 Monoids	196
	14.2 Foldables	200
	14.3 Traversables	206
	14.4 Chapter remarks	210
	14.5 Exercises	210
<b>15</b>	<b>Lazy evaluation</b>	<b>212</b>
	15.1 Introduction	212
	15.2 Evaluation strategies	213
	15.3 Termination	216
	15.4 Number of reductions	217
	15.5 Infinite structures	219
	15.6 Modular programming	220
	15.7 Strict application	223
	15.8 Chapter remarks	226
	15.9 Exercises	226



<b>16</b>	<b>Reasoning about programs</b>	228
	16.1 Equational reasoning	228
	16.2 Reasoning about Haskell	229
	16.3 Simple examples	230
	16.4 Induction on numbers	231
	16.5 Induction on lists	234
	16.6 Making append vanish	238
	16.7 Compiler correctness	241
	16.8 Chapter remarks	246
	16.9 Exercises	246
<b>17</b>	<b>Calculating compilers</b>	249
	17.1 Introduction	249
	17.2 Syntax and semantics	249
	17.3 Adding a stack	250
	17.4 Adding a continuation	252
	17.5 Defunctionalising	254
	17.6 Combining the steps	257
	17.7 Chapter remarks	261
	17.8 Exercises	261
<b>Appendix A</b>	<b>Selected solutions</b>	263
	A.1 Introduction	263
	A.2 First steps	264
	A.3 Types and classes	265
	A.4 Defining functions	266
	A.5 List comprehensions	267
	A.6 Recursive functions	267
	A.7 Higher-order functions	268
	A.8 Declaring types and classes	269
	A.9 The countdown problem	270
	A.10 Interactive programming	270
	A.11 Unbeatable tic-tac-toe	271
	A.12 Monads and more	272
	A.13 Monadic parsing	273
	A.14 Foldables and friends	274
	A.15 Lazy evaluation	275
	A.16 Reasoning about programs	276
	A.17 Calculating compilers	279
<b>Appendix B</b>	<b>Standard prelude</b>	280
	B.1 Basic classes	280
	B.2 Booleans	281
	B.3 Characters	282

B.4	Strings	283
B.5	Numbers	283
B.6	Tuples	284
B.7	Maybe	284
B.8	Lists	285
B.9	Functions	287
B.10	Input/output	287
B.11	Functors	288
B.12	Applicatives	289
B.13	Monads	290
B.14	Alternatives	290
B.15	MonadPlus	291
B.16	Monoids	292
B.17	Foldables	294
B.18	Traversables	297
	<i>Bibliography</i>	298
	<i>Index</i>	300

## Foreword

It is nearly a century ago that Alonzo Church introduced the lambda calculus, and over half a century ago that John McCarthy introduced Lisp, the world's second oldest programming language and the first functional language based on the lambda calculus. By now, every major programming language including JavaScript, C++, Swift, Python, PHP, Visual Basic, Java, . . . has support for lambda expressions or anonymous higher-order functions.

As with any idea that becomes mainstream, inevitably the underlying foundations and principles get watered down or forgotten. Lisp allowed mutation, yet today many confuse functions as first-class citizens with immutability. At the same time, other effects such as exceptions, reflection, communication with the outside world, and concurrency go unmentioned. Adding recursion in the form of feedback-loops to pure combinational circuits lets us implement mutable state via flip-flops. Similarly, using one effect such as concurrency or input/output we can simulate other effects such as mutability. John Hughes famously stated in his classic paper *Why Functional Programming Matters* that we cannot make a language more powerful by eliminating features. To that, we add that often we cannot even make a language less powerful by removing features. In this book, Graham demonstrates convincingly that the true value of functional programming lies in leveraging first-class functions to achieve compositionality and equational reasoning. Or in Graham's own words, "functional programming can be viewed as a style of programming in which the basic method of computation is the application of functions to arguments". These functions do not necessarily have to be pure or statically typed in order to realise the simplicity, elegance, and conciseness of expression that we get from the functional style.

While you can code like a functional hacker in a plethora of languages, a semantically pure and lazy, and syntactically lean and terse language such as Haskell is still the best way to learn how to think like a fundamentalist. Based upon decades of teaching experience, and backed by an impressive stream of research papers, in this book Graham gently guides us through the whole gambit of key functional programming concepts such as higher-order functions, recursion, list comprehensions, algebraic datatypes and pattern matching. The book does not shy away from more advanced concepts. If you are still confused by the n-th blog post that attempts to explain monads, you are in the right place. Gently starting with the IO monad, Graham progresses from functors to applicatives using many concrete examples. By the time he arrives at monads, every reader will feel that they themselves could have come up with the concept of a monad as a generic pattern for composing functions with effects. The chapter on monadic

parsers brings everything together in a compelling use-case of parsing arithmetic expressions in the implementation of a simple calculator.

This new edition not only adds many more concrete examples of concepts introduced throughout the book, it also introduces the novel Haskell concepts of foldable and traversable types. Readers familiar with object-oriented languages routinely use iterables and visitors to enumerate over all values in a container, or respectively to traverse complex data structures. Haskell’s higher-kinded type classes allow for a very concise and abstract treatment of these concepts by means of the Foldable and Traversable classes. Last but not least, the final chapters of the book give an in-depth overview of lazy evaluation and equational reasoning to prove and derive programs. The capstone chapter on calculating compilers especially appeals to me because it touches a topic that has had my keen interest for many decades, ever since my own PhD thesis on the same topic.

While there are plenty of alternative textbooks on Haskell in particular and functional programming in general, Graham’s book is unique amongst all of these in that it uses Haskell simply as a tool for thought, and never attempts to sell Haskell or functional programming as a silver bullet that magically solves all programming problems. It focuses on elegant and concise expression of intent and thus makes a strong case of how pure and lazy functional programming is an intelligible medium for efficiently reasoning about algorithms at a high level of abstraction. The skills you acquire by studying this book will make you a much better programmer no matter what language you use to actually program in. In the past decade, using the first edition of this book I have taught many tens of thousands of students how to juggle with code. With this new edition, I am looking forward to extending this streak for at least another 10 years.

Erik Meijer

## Preface

### What is this book?

Haskell is a purely functional language that allows programmers to rapidly develop software that is clear, concise and correct. The book is aimed at a broad spectrum of readers who are interested in learning the language, including professional programmers, university students and high-school students. However, no programming experience is required or assumed, and all concepts are explained from first principles with the aid of carefully chosen examples and exercises. Most of the material in the book should be accessible to anyone over the age of around sixteen with a reasonable aptitude for scientific ideas.

### How is it structured?

The book is divided into two parts. Part I introduces the basic concepts of pure programming in Haskell and is structured around the core features of the language, such as types, functions, list comprehensions, recursion and higher-order functions. Part II covers impure programming and a range of more advanced topics, such as monads, parsing, foldable types, lazy evaluation and reasoning about programs. The book contains many extended programming examples, and each chapter includes suggestions for further reading and a series of exercises. The appendices provide solutions to selected exercises, and a summary of some of the most commonly used definitions from the Haskell standard prelude.

### What is its approach?

The book aims to teach the key concepts of Haskell in a clean and simple manner. As this is a textbook rather than a reference manual we do not attempt to cover all aspects of the language and its libraries, and we sometimes choose to define functions from first principles rather than using library functions. As the book progresses the level of generality that is used is gradually increased. For example, in the beginning most of the functions that are used are specialised to simple types, and later on we see how many functions can be generalised to larger classes of types by exploiting particular features of Haskell.

### How should it be read?

The basic material in part I can potentially be worked through fairly quickly, particularly for those with some prior programming experience, but additional time and effort may be required to absorb some of material in part II. Readers are recommended to work through all the material in part I, and then select

appropriate material from part II depending on their own interests. It is vital to write Haskell code for yourself as you go along, as you can't learn to program just by reading. Try out the examples from each chapter as you proceed, and solve the exercises for each chapter before checking the solutions.

### What's new in this edition?

The book is an extensively revised and expanded version of the first edition. It has been extended with new chapters that cover more advanced aspects of Haskell, new examples and exercises to further reinforce the concepts being introduced, and solutions to selected exercises. The remaining material has been completely reworked in response to changes in the language and feedback from readers. The new edition uses the Glasgow Haskell Compiler (GHC), and is fully compatible with the latest version of the language, including recent changes concerning applicative, monadic, foldable and traversable types.

### How can it be used for teaching?

An introductory course might cover all of part I and a few selected topics from part II; my first-year course covers chapters 1–9, 10 and 15. An advanced course might start with a refresher of part I, and cover a selection of more advanced topics from part II; my second-year course focuses on chapters 12 and 16, and is taught interactively on the board. The website for the book provides a range of supporting materials, including PowerPoint slides and Haskell code for the extended examples. Instructors can obtain a large collection of exams and solutions based on material in the book from [solutions@cambridge.org](mailto:solutions@cambridge.org).

### Acknowledgements

I am grateful to the University of Nottingham for providing a sabbatical to produce this new edition; Thorsten Altenkirch, Venanzio Capretta, Henrik Nilsson and other members of the FP lab for our many enjoyable discussions; Iván Pérez Domínguez for useful comments on a number of chapters; the students and tutors on all of my Haskell courses for their feedback; Clare Dennison, David Tranah and Abigail Walkington at CUP for their editorial work; the GHC team for producing such a great compiler; and finally, Catherine and Ian Hutton for getting me started in computing all those years ago.

Many thanks also to Ki Yung Ahn, Bob Davison, Philip Hölzenspies and Neil Mitchell for providing detailed comments on the first edition, and to the following for pointing our errors and typos: Paul Brown, Sergio Queiroz de Medeiros, David Duke, Robert Fabian, Ben Fleis, Robert Furber, Andrew Kish, Tomoyas Kobayashi, Florian Larysch, Carlos Oroz, Douglas Philips, Bruce Turner, Gregor Ulm, Marco Valtorta and Kazu Yamamoto. All of these comments have been taken into account when preparing the new edition.

Graham Hutton