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
For Annette, Callum and Tom
# Contents

`Foreword` | page xiii  
`Preface` | xv  

## Part I Basic Concepts  

1 **Introduction**  
1.1 Functions  
1.2 Functional programming  
1.3 Features of Haskell  
1.4 Historical background  
1.5 A taste of Haskell  
1.6 Chapter remarks  
1.7 Exercises  

2 **First steps**  
2.1 Glasgow Haskell Compiler  
2.2 Installing and starting  
2.3 Standard prelude  
2.4 Function application  
2.5 Haskell scripts  
2.6 Chapter remarks  
2.7 Exercises  

3 **Types and classes**  
3.1 Basic concepts  
3.2 Basic types  
3.3 List types  
3.4 Tuple types  
3.5 Function types  
3.6 Curried functions  
3.7 Polymorphic types  
3.8 Overloaded types  
3.9 Basic classes  
3.10 Chapter remarks
## Contents

3.11 Exercises 36

4 Defining functions 38
  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

5 List comprehensions 47
  5.1 Basic concepts 47
  5.2 Guards 48
  5.3 The `zip` function 50
  5.4 String comprehensions 51
  5.5 The Caesar cipher 52
  5.6 Chapter remarks 56
  5.7 Exercises 57

6 Recursive functions 59
  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

7 Higher-order functions 73
  7.1 Basic concepts 73
  7.2 Processing lists 74
  7.3 The `foldr` function 76
  7.4 The `foldl` 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

8 Declaring types and classes 92
  8.1 Type declarations 92
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>Data declarations</td>
<td>93</td>
</tr>
<tr>
<td>8.3</td>
<td>Newtype declarations</td>
<td>95</td>
</tr>
<tr>
<td>8.4</td>
<td>Recursive types</td>
<td>96</td>
</tr>
<tr>
<td>8.5</td>
<td>Class and instance declarations</td>
<td>99</td>
</tr>
<tr>
<td>8.6</td>
<td>Tautology checker</td>
<td>101</td>
</tr>
<tr>
<td>8.7</td>
<td>Abstract machine</td>
<td>106</td>
</tr>
<tr>
<td>8.8</td>
<td>Chapter remarks</td>
<td>108</td>
</tr>
<tr>
<td>8.9</td>
<td>Exercises</td>
<td>109</td>
</tr>
<tr>
<td>9</td>
<td>The countdown problem</td>
<td>111</td>
</tr>
<tr>
<td>9.1</td>
<td>Introduction</td>
<td>111</td>
</tr>
<tr>
<td>9.2</td>
<td>Arithmetic operators</td>
<td>112</td>
</tr>
<tr>
<td>9.3</td>
<td>Numeric expressions</td>
<td>113</td>
</tr>
<tr>
<td>9.4</td>
<td>Combinatorial functions</td>
<td>114</td>
</tr>
<tr>
<td>9.5</td>
<td>Formalising the problem</td>
<td>115</td>
</tr>
<tr>
<td>9.6</td>
<td>Brute force solution</td>
<td>115</td>
</tr>
<tr>
<td>9.7</td>
<td>Performance testing</td>
<td>116</td>
</tr>
<tr>
<td>9.8</td>
<td>Combining generation and evaluation</td>
<td>117</td>
</tr>
<tr>
<td>9.9</td>
<td>Exploiting algebraic properties</td>
<td>118</td>
</tr>
<tr>
<td>9.10</td>
<td>Chapter remarks</td>
<td>119</td>
</tr>
<tr>
<td>9.11</td>
<td>Exercises</td>
<td>120</td>
</tr>
<tr>
<td>10</td>
<td>Interactive programming</td>
<td>121</td>
</tr>
<tr>
<td>10.1</td>
<td>The problem</td>
<td>122</td>
</tr>
<tr>
<td>10.2</td>
<td>The solution</td>
<td>123</td>
</tr>
<tr>
<td>10.3</td>
<td>Basic actions</td>
<td>124</td>
</tr>
<tr>
<td>10.4</td>
<td>Sequencing</td>
<td>125</td>
</tr>
<tr>
<td>10.5</td>
<td>Derived primitives</td>
<td>126</td>
</tr>
<tr>
<td>10.6</td>
<td>Hangman</td>
<td>127</td>
</tr>
<tr>
<td>10.7</td>
<td>Nim</td>
<td>128</td>
</tr>
<tr>
<td>10.8</td>
<td>Life</td>
<td>129</td>
</tr>
<tr>
<td>10.9</td>
<td>Chapter remarks</td>
<td>133</td>
</tr>
<tr>
<td>10.10</td>
<td>Exercises</td>
<td>137</td>
</tr>
<tr>
<td>11</td>
<td>Unbeatable tic-tac-toe</td>
<td>139</td>
</tr>
<tr>
<td>11.1</td>
<td>Introduction</td>
<td>139</td>
</tr>
<tr>
<td>11.2</td>
<td>Basic declarations</td>
<td>140</td>
</tr>
<tr>
<td>11.3</td>
<td>Grid utilities</td>
<td>141</td>
</tr>
<tr>
<td>11.4</td>
<td>Displaying a grid</td>
<td>142</td>
</tr>
<tr>
<td>11.5</td>
<td>Making a move</td>
<td>143</td>
</tr>
<tr>
<td>11.6</td>
<td>Reading a number</td>
<td>144</td>
</tr>
<tr>
<td>11.7</td>
<td>Human vs human</td>
<td>144</td>
</tr>
</tbody>
</table>
## Contents

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  

### 12 Monads and more

12.1 Functors 153  
12.2 Applicatives 157  
12.3 Monads 164  
12.4 Chapter remarks 174  
12.5 Exercises 175  

### 13 Monadic parsing

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  

### 14 Foldables and friends

14.1 Monoids 196  
14.2 Foldables 200  
14.3 Traversables 206  
14.4 Chapter remarks 210  
14.5 Exercises 210  

### 15 Lazy evaluation

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
Contents

16  Reasoning about programs  
16.1  Equational reasoning  
16.2  Reasoning about Haskell  
16.3  Simple examples  
16.4  Induction on numbers  
16.5  Induction on lists  
16.6  Making append vanish  
16.7  Compiler correctness  
16.8  Chapter remarks  
16.9  Exercises  

17  Calculating compilers  
17.1  Introduction  
17.2  Syntax and semantics  
17.3  Adding a stack  
17.4  Adding a continuation  
17.5  Defunctionalising  
17.6  Combining the steps  
17.7  Chapter remarks  
17.8  Exercises  

Appendix A  Selected solutions  
A.1  Introduction  
A.2  First steps  
A.3  Types and classes  
A.4  Defining functions  
A.5  List comprehensions  
A.6  Recursive functions  
A.7  Higher-order functions  
A.8  Declaring types and classes  
A.9  The countdown problem  
A.10  Interactive programming  
A.11  Unbeatable tic-tac-toe  
A.12  Monads and more  
A.13  Monadic parsing  
A.14  Foldables and friends  
A.15  Lazy evaluation  
A.16  Reasoning about programs  
A.17  Calculating compilers  

Appendix B  Standard prelude  
B.1  Basic classes  
B.2  Booleans  
B.3  Characters  

© in this web service Cambridge University Press  www.cambridge.org
### Contents

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

*Bibliography* 298

*Index* 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-kindred 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 ex-
tended examples. Instructors can obtain a large collection of exams and solutions
based on material in the book from solutions@cambridge.org.

Acknowledgements

I am grateful to the University of Nottingham for providing a sabbatical to pro-
duce 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 pro-
ducing 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 out 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