Maps, Monads, Monoids, and More!

It’s all in the name: Learn You a Haskell for Great Good! is a hilarious, illustrated guide to this complex functional language. Packed with the author’s original artwork, pop culture references, and, most importantly, useful example code, this book teaches functional fundamentals in a way you never thought possible.

You’ll start with the kid stuff: basic syntax, recursion, types, and type classes. Then once you’ve got the basics down, the real black-belt master class begins: you’ll learn to use applicative functors, monads, zippers, and all the other mythical Haskell constructs you’ve only read about in storybooks.

As you work your way through the author’s imaginative (and occasionally insane) examples, you’ll learn to:

- Laugh in the face of side effects as you wield purely functional programming techniques
- Use the magic of Haskell’s “laziness” to play with infinite sets of data
- Organize your programs by creating your own types, type classes, and modules
- Use Haskell’s elegant input/output system to share the genius of your programs with the outside world

Short of eating the author’s brain, you will not find a better way to learn this powerful language than reading Learn You a Haskell for Great Good!

About the Author

Miran Lipovača is a computer science student in Ljubljana, Slovenia. In addition to his passion for Haskell, he enjoys boxing, playing bass guitar, and, of course, drawing. He has a fascination with dancing skeletons and the number 71, and when he walks through automatic doors, he pretends that he’s actually opening them with his mind.
LEARN YOU A HASKELL FOR GREAT GOOD!
BRIEF CONTENTS

Introduction ................................................................. xv
Chapter 1: Starting Out ......................................................... 1
Chapter 2: Believe the Type ..................................................... 23
Chapter 3: Syntax in Functions ............................................... 35
Chapter 4: Hello Recursion! .................................................... 51
Chapter 5: Higher-Order Functions ......................................... 59
Chapter 6: Modules ............................................................. 87
Chapter 7: Making Our Own Types and Type Classes ..................... 109
Chapter 8: Input and Output .................................................. 153
Chapter 9: More Input and More Output .................................... 169
Chapter 10: Functionally Solving Problems .................................. 203
Chapter 11: Applicative Functors ............................................ 217
Chapter 12: Monoids ........................................................... 243
Chapter 13: A Fistful of Monads .............................................. 267
Chapter 14: For a Few Monads More ......................................... 297
Chapter 15: Zippers .............................................................. 343
Index ............................................................................. 363
## CONTENTS IN DETAIL

### INTRODUCTION  xv
- So, What's Haskell?  xv
- What You Need to Dive In  xvii
- Acknowledgments  xviii

### 1 STARTING OUT  1
- Calling Functions  3
- Baby’s First Functions  5
- An Intro to Lists  7
  - Concatenation  8
  - Accessing List Elements  9
  - Lists Inside Lists  9
  - Comparing Lists  9
  - More List Operations  10
- Texas Ranges  13
- I’m a List Comprehension  15
- Tuples  18
  - Using Tuples  19
  - Using Pairs  20
  - Finding the Right Triangle  21

### 2 BELIEVE THE TYPE  23
- Explicit Type Declaration  24
- Common Haskell Types  25
- Type Variables  26
- Type Classes 101  27
  - The Eq Type Class  28
  - The Ord Type Class  28
  - The Show Type Class  29
  - The Read Type Class  29
  - The Enum Type Class  31
  - The Bounded Type Class  31
  - The Num Type Class  32
  - The Floating Type Class  32
  - The Integral Type Class  33
- Some Final Notes on Type Classes  33
3 SYNTAX IN FUNCTIONS

Pattern Matching ................................................................. 35
  Pattern Matching with Tuples ........................................... 37
  Pattern Matching with Lists and List Comprehensions .......... 38
  As-patterns ................................................................. 40
Guards, Guards! ............................................................... 40
where?! ........................................................................ 42
  where’s Scope ............................................................ 44
  Pattern Matching with where ......................................... 44
  Functions in where Blocks ............................................ 45
let It Be ..................................................................... 45
  let in List Comprehensions ........................................... 47
  let in GHCi .............................................................. 47
case Expressions ............................................................ 48

4 HELLO RECURSION!

Maximum Awesome .......................................................... 51
A Few More Recursive Functions ......................................... 52
  replicate ................................................................. 53
  take ....................................................................... 53
  reverse ................................................................. 54
  repeat ................................................................. 55
  zip ................................................................. 55
  elem ................................................................. 56
Quick, Sort! ...................................................................... 56
  The Algorithm ........................................................ 56
  The Code .............................................................. 57
Thinking Recursively ......................................................... 58

5 HIGHER-ORDER FUNCTIONS

Curried Functions ............................................................. 59
  Sections ................................................................. 62
  Printing Functions ..................................................... 63
Some Higher-Orderism Is in Order ....................................... 63
  Implementing zipWith ............................................... 64
  Implementing flip .................................................... 65
The Functional Programmer’s Toolbox .................................. 66
  The map Function ................................................... 66
  The filter Function ................................................ 67
  More Examples of map and filter .................................. 68
  Mapping Functions with Multiple Parameters .................. 70
Lambdas ................................................................. 71
## Contents

### 6 modules

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Importing Modules</td>
<td>88</td>
</tr>
<tr>
<td>Solving Problems with Module Functions</td>
<td>90</td>
</tr>
<tr>
<td>Counting Words</td>
<td>90</td>
</tr>
<tr>
<td>Needle in the Haystack</td>
<td>91</td>
</tr>
<tr>
<td>Caesar Cipher Salad</td>
<td>92</td>
</tr>
<tr>
<td>On Strict Left Folds</td>
<td>94</td>
</tr>
<tr>
<td>Let's Find Some Cool Numbers</td>
<td>95</td>
</tr>
<tr>
<td>Mapping Keys to Values</td>
<td>98</td>
</tr>
<tr>
<td>Almost As Good: Association Lists</td>
<td>98</td>
</tr>
<tr>
<td>Enter Data.Map</td>
<td>100</td>
</tr>
<tr>
<td>Making Our Own Modules</td>
<td>104</td>
</tr>
<tr>
<td>A Geometry Module</td>
<td>104</td>
</tr>
<tr>
<td>Hierarchical Modules</td>
<td>106</td>
</tr>
</tbody>
</table>

### 7 making our own types and type classes

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Defining a New Data Type</td>
<td>109</td>
</tr>
<tr>
<td>Shaping Up</td>
<td>110</td>
</tr>
<tr>
<td>Improving Shape with the Point Data Type</td>
<td>112</td>
</tr>
<tr>
<td>Exporting Our Shapes in a Module</td>
<td>113</td>
</tr>
<tr>
<td>Record Syntax</td>
<td>114</td>
</tr>
<tr>
<td>Type Parameters</td>
<td>117</td>
</tr>
<tr>
<td>Should We Parameterize Our Car?</td>
<td>119</td>
</tr>
<tr>
<td>Vector von Doom</td>
<td>121</td>
</tr>
<tr>
<td>Derived Instances</td>
<td>122</td>
</tr>
<tr>
<td>Equating People</td>
<td>123</td>
</tr>
<tr>
<td>Show Me How to Read</td>
<td>124</td>
</tr>
<tr>
<td>Order in the Court!</td>
<td>125</td>
</tr>
<tr>
<td>Any Day of the Week</td>
<td>126</td>
</tr>
</tbody>
</table>
## Contents in Detail

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Functionally Solving Problems</td>
<td>203</td>
</tr>
<tr>
<td></td>
<td>Reverse Polish Notation Calculator</td>
<td>203</td>
</tr>
<tr>
<td></td>
<td>Calculating RPN Expressions</td>
<td>204</td>
</tr>
<tr>
<td></td>
<td>Writing an RPN Function</td>
<td>205</td>
</tr>
<tr>
<td></td>
<td>Adding More Operators</td>
<td>207</td>
</tr>
<tr>
<td></td>
<td>Heathrow to London</td>
<td>208</td>
</tr>
<tr>
<td></td>
<td>Calculating the Quickest Path</td>
<td>209</td>
</tr>
<tr>
<td></td>
<td>Representing the Road System in Haskell</td>
<td>211</td>
</tr>
<tr>
<td></td>
<td>Writing the Optimal Path Function</td>
<td>212</td>
</tr>
<tr>
<td></td>
<td>Getting a Road System from the Input</td>
<td>215</td>
</tr>
<tr>
<td>11</td>
<td>Applicative Functors</td>
<td>217</td>
</tr>
<tr>
<td></td>
<td>Functors Redux</td>
<td>218</td>
</tr>
<tr>
<td></td>
<td>I/O Actions As Functors</td>
<td>218</td>
</tr>
<tr>
<td></td>
<td>Functions As Functors</td>
<td>220</td>
</tr>
<tr>
<td></td>
<td>Functor Laws</td>
<td>223</td>
</tr>
<tr>
<td></td>
<td>Law 1</td>
<td>223</td>
</tr>
<tr>
<td></td>
<td>Law 2</td>
<td>224</td>
</tr>
<tr>
<td></td>
<td>Breaking the Law</td>
<td>225</td>
</tr>
<tr>
<td></td>
<td>Using Applicative Functors</td>
<td>227</td>
</tr>
<tr>
<td></td>
<td>Say Hello to Applicative</td>
<td>228</td>
</tr>
<tr>
<td></td>
<td>Maybe the Applicative Functor</td>
<td>229</td>
</tr>
<tr>
<td></td>
<td>The Applicative Style</td>
<td>230</td>
</tr>
</tbody>
</table>
14
FOR A FEW MONADS MORE
Writer? I Hardly Knew Her! ................................................................. 298
  Monoids to the Rescue ................................................................. 299
  The Writer Type ................................................................. 300
Using do Notation with Writer .......................................................... 302
Adding Logging to Programs ......................................................... 303
inefficient List Construction .......................................................... 304
Using Difference Lists ..................................................................... 305
Comparing Performance ..................................................................... 306
Reader? Ugh, Not This Joke Again .................................................... 307
Functions As Monads ....................................................................... 308
The Reader Monad ........................................................................... 309
Tasteful Stateful Computations ....................................................... 310
  Stateful Computations .................................................................... 311
  Stacks and Stones ......................................................................... 312
Stateful Computations ..................................................................... 313
  The State Monad ............................................................................ 314
  Getting and Setting State ............................................................... 315
Randomness and the State Monad .................................................... 316
Error Error on the Wall ..................................................................... 317
Some Useful Monadic Functions ....................................................... 318
  liftM and Friends ......................................................................... 319
  The join Function ......................................................................... 320
  filterM ......................................................................................... 321
  foldM ......................................................................................... 322
Making a Safe RPN Calculator ......................................................... 323
Comparing Performance .................................................................... 324
Using Difference Lists ..................................................................... 325
Using do Notation with Writer ......................................................... 326
The Writer Type ............................................................................... 327
Monoids to the Rescue ..................................................................... 328
The join Function ............................................................................ 329
Using Difference Lists ..................................................................... 330
Making a Safe RPN Calculator ......................................................... 331
Making Monads ............................................................................... 332

15
ZIPPERS
Taking a Walk .................................................................................... 333
  A Trail of Breadcrumbs .................................................................... 334
  Going Back Up .................................................................................. 335
Manipulating Trees Under Focus .................................................... 336
  Going Straight to the Top, Where the Air Is Fresh and Clean! .......... 337
Focusing on Lists ............................................................................. 338
A Very Simple Filesystem ................................................................. 339
  Making a Zipper for Our Filesystem ............................................. 340
  Manipulating a Filesystem ............................................................ 341
Watch Your Step ............................................................................... 342
Thanks for Reading! ......................................................................... 343

INDEX
363
Haskell is fun, and that’s what it’s all about!

This book is aimed at people who have experience programming in imperative languages—such as C++, Java, and Python—and now want to try out Haskell. But even if you don’t have any significant programming experience, I’ll bet a smart person like you will be able to follow along and learn Haskell.

My first reaction to Haskell was that the language was just too weird. But after getting over that initial hurdle, it was smooth sailing. Even if Haskell seems strange to you at first, don’t give up. Learning Haskell is almost like learning to program for the first time all over again. It’s fun, and it forces you to think differently.

**NOTE**  
If you ever get really stuck, the IRC channel #haskell on the freenode network is a great place to ask questions. The people there tend to be nice, patient, and understanding. They’re a great resource for Haskell newbies.

**So, What’s Haskell?**

Haskell is a *purely functional* programming language.

In *imperative* programming languages, you give the computer a sequence of tasks, which it then executes. While executing them, the computer can change state. For instance, you can set the variable `a` to 5 and then do some stuff that might change the value of `a`. There are also flow-control structures for executing instructions several times, such as `for` and `while` loops.
Purely functional programming is different. You don’t tell the computer what to do—you tell it *what stuff is*. For instance, you can tell the computer that the factorial of a number is the product of every integer from 1 to that number or that the sum of a list of numbers is the first number plus the sum of the remaining numbers. You can express both of these operations as *functions*.

In functional programming, you *can’t* set a variable to one value and then set it to something else later on. If you say $a$ is 5, you can’t just change your mind and say it’s something else. After all, you said it was 5. (What are you, some kind of liar?)

In purely functional languages, a function has no *side effects*. The only thing a function can do is calculate something and return the result. At first, this seems limiting, but it actually has some very nice consequences. If a function is called twice with the same parameters, it’s guaranteed to return the same result both times. This property is called *referential transparency*. It lets the programmer easily deduce (and even prove) that a function is correct. You can then build more complex functions by gluing these simple functions together.

Haskell is *lazy*. This means that unless specifically told otherwise, Haskell won’t execute functions until it needs to show you a result. This is made possible by referential transparency. If you know that the result of a function depends only on the parameters that function is given, it doesn’t matter when you actually calculate the result of the function. Haskell, being a lazy language, takes advantage of this fact and defers actually computing results for as long as possible. Once you want your results to be displayed, Haskell will do just the bare minimum computation required to display them. Laziness also allows you to make seemingly infinite data structures, because only the parts of the data structures that you choose to display will actually be computed.

Let’s look at an example of Haskell’s laziness. Say you have a list of numbers, $xs = [1,2,3,4,5,6,7,8]$, and a function called `doubleMe` that doubles every element and returns the result as a new list. If you want to multiply your list by 8, your code might look something like this:

```haskell
doubleMe(doubleMe(doubleMe(xs)))
```
An imperative language would probably pass through the list once, make a copy, and then return it. It would then pass through the list another two times, making copies each time, and return the result.

In a lazy language, calling `doubleMe` on a list without forcing it to show you the result just makes the program tell you, “Yeah yeah, I’ll do it later!” Once you want to see the result, the first `doubleMe` calls the second one and says it wants the result immediately. Then the second one says the same thing to the third one, and the third one reluctantly gives back a doubled 1, which is 2. The second `doubleMe` receives that and returns 4 to the first one. The first `doubleMe` then doubles this result and tells you that the first element in the final resulting list is 8. Because of Haskell’s laziness, the `doubleMe` calls pass through the list just once, and only when you really need that to happen.

Haskell is *statically typed*. This means that when you compile your program, the compiler knows which piece of code is a number, which is a string, and so on. Static typing means that a lot of possible errors can be caught at compile time. If you try to add together a number and a string, for example, the compiler will whine at you.

Haskell uses a very good type system that has *type inference*. This means that you don’t need to explicitly label every piece of code with a type, because Haskell’s type system can intelligently figure it out. For example, if you say `a = 5 + 4`, you don’t need to tell Haskell that `a` is a number—it can figure that out by itself. Type inference makes it easier for you to write code that’s more general. If you write a function that takes two parameters and adds them together, but you don’t explicitly state their type, the function will work on any two parameters that act like numbers.

Haskell is *elegant and concise*. Because it uses a lot of high-level concepts, Haskell programs are usually shorter than their imperative equivalents. Shorter programs are easier to maintain and have fewer bugs.

Haskell was made by some really smart guys (with PhDs). Work on Haskell began in 1987 when a committee of researchers got together to design a kick-ass language. The Haskell Report, which defines a stable version of the language, was published in 1999.

## What You Need to Dive In

In short, to get started with Haskell, you need a text editor and a Haskell compiler. You probably already have your favorite text editor installed, so we won’t waste time on that. The most popular Haskell compiler is the Glasgow Haskell Compiler (GHC), which we will be using throughout this book.

The best way to get what you need is to download the *Haskell Platform*. The Haskell Platform includes not only the GHC compiler but also a bunch of useful Haskell libraries! To get the Haskell Platform for your system, go to
http://hackage.haskell.org/platform/ and follow the instructions for your operating system.

GHC can compile Haskell scripts (usually with an .hs extension), and it also has an interactive mode. From there, you can load functions from scripts and then call them directly to see immediate results. Especially when you’re learning, it’s much easier to use the interactive mode than it is to compile and run your code every time you make a change.

Once you’ve installed the Haskell Platform, open a new terminal window, assuming you’re on a Linux or Mac OS X system. If your operating system of choice is Windows, go to the command prompt. Once there, type ghci and press ENTER to start the interactive mode. (If your system fails to find the GHCi program, you can try rebooting your computer.)

If you’ve defined some functions in a script—for example, myfunctions.hs—you can load these functions into GHCi by typing :l myfunctions. (Make sure that myfunctions.hs is in the same folder from which you started GHCi.)

If you change the .hs script, run :l myfunctions to load the file again or run :r, which reloads the current script. My usual workflow is to define some functions in an .hs file, load it into GHCi, mess around with it, change the file, and repeat. This is what we’ll be doing in this book.

Acknowledgments

Thanks to everyone who sent in corrections, suggestions, and words of encouragement. Also thanks to Keith, Sam, and Marilyn for making me look like a real writer.