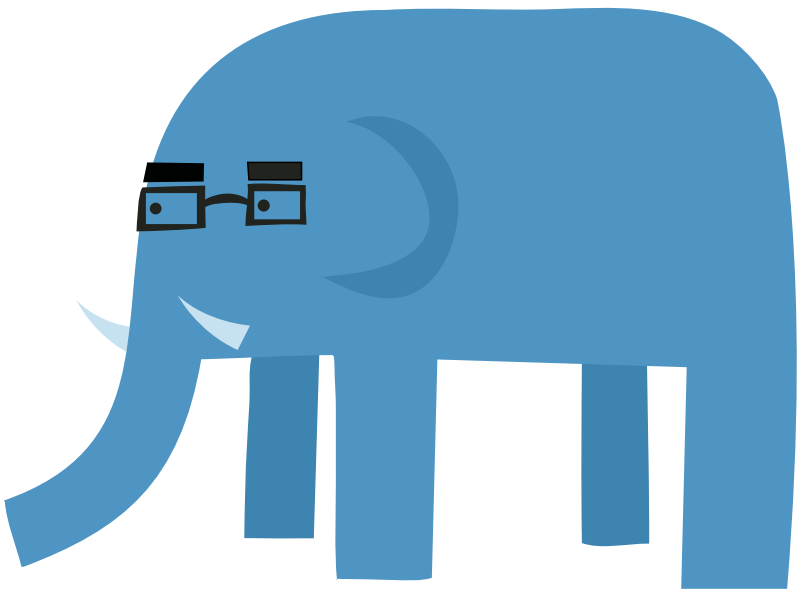
# Learn You a Haskell for Great Good!

## A Beginner's Guide



Miran Lipovača

# LEARN YOU A HASKELL FOR GREAT GOOD!

# Learn You a Haskell for Great Good!

## A Beginner's Guide

Miran Lipovača


San Francisco

15 14 13 12 11 1 2 3 4 5 6 7 8 9

ISBN-10: 1-59327-283-9
ISBN-13: 978-1-59327-283-8

Publisher: William Pollock
Production Editors: Ansel Staton and Serena Yang
Cover and Interior Design: Octopod Studios
Developmental Editor: Keith Fancher
Technical Reviewer: Samuel Hughes
Copyeditor: Marilyn Smith
Compositor: Alison Law
Proofreader: Ellen Brink
Indexer: Valerie Haynes Perry

For information on book distributors or translations, please contact No Starch Press, Inc. directly:

No Starch Press, Inc.
38 Ringold Street, San Francisco, CA 94103
phone: 415.863.9900; fax: 415.863.9950; info@nostarch.com; www.nostarch.com

*Library of Congress Cataloging-in-Publication Data*

```
Lipovaca, Miran.
 Learn you a Haskell for great good! : a beginner's guide / by Miran Lipovaca.
      p. cm.
 ISBN-13: 978-1-59327-283-8
 ISBN-10: 1-59327-283-9
 1.  Haskell (Computer program language)  I. Title.
 QA76.73.H37L69 2012
 005.13'3-dc22
                                                          2011000790
```

# 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** **35**

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!** **51**

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

**5**
**HIGHER-ORDER FUNCTIONS** **59**

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

I Fold You So ..... 73
Left Folds with foldl ..... 74
Right Folds with foldr ..... 75
The foldl and foldr1 Functions ..... 76
Some Fold Examples ..... 76
Another Way to Look at Folds ..... 77
Folding Infinite Lists ..... 78
Scans ..... 79
Function Application with $ ..... 80
Function Composition ..... 82
Function Composition with Multiple Parameters ..... 83
Point-Free Style ..... 84

**6**
**MODULES** **87**

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

**7**
**MAKING OUR OWN TYPES AND TYPE CLASSES** **109**

Defining a New Data Type ..... 109
Shaping Up ..... 110
Improving Shape with the Point Data Type ..... 112
Exporting Our Shapes in a Module ..... 113
Record Syntax ..... 114
Type Parameters ..... 117
Should We Parameterize Our Car? ..... 119
Vector von Doom ..... 121
Derived Instances ..... 122
Equating People ..... 123
Show Me How to Read ..... 124
Order in the Court! ..... 125
Any Day of the Week ..... 126

Type Synonyms ..... 127
Making Our Phonebook Prettier ..... 128
Parameterizing Type Synonyms ..... 129
Go Left, Then Right ..... 130
Recursive Data Structures ..... 132
Improving Our List ..... 133
Let's Plant a Tree ..... 135
Type Classes 102 ..... 138
Inside the Eq Type Class ..... 138
A Traffic Light Data Type ..... 139
Subclassing ..... 140
Parameterized Types As Instances of Type Classes ..... 141
A Yes-No Type Class ..... 143
The Functor Type Class ..... 146
Maybe As a Functor ..... 147
Trees Are Functors, Too ..... 148
Either a As a Functor ..... 149
Kinds and Some Type-Foo ..... 150

**8**
**INPUT AND OUTPUT** **153**

Separating the Pure from the Impure ..... 153
Hello, World! ..... 154
Gluing I/O Actions Together ..... 156
Using let Inside I/O Actions ..... 158
Putting It in Reverse ..... 159
Some Useful I/O Functions ..... 161
putStr ..... 161
putChar ..... 162
print ..... 162
when ..... 163
sequence ..... 164
mapM ..... 165
forever ..... 165
forM ..... 166
I/O Action Review ..... 167

**9**
**MORE INPUT AND MORE OUTPUT** **169**

Files and Streams ..... 169
Input Redirection ..... 170
Getting Strings from Input Streams ..... 171
Transforming Input ..... 173

Reading and Writing Files ... 175
Using the withFile Function ... 177
It's Bracket Time ... 178
Grab the Handles! ... 179
To-Do Lists ... 180
Deleting Items ... 181
Cleaning Up ... 183
Command-Line Arguments ... 184
More Fun with To-Do Lists ... 185
A Multitasking Task List ... 186
Dealing with Bad Input ... 190
Randomness ... 190
Tossing a Coin ... 193
More Random Functions ... 194
Randomness and I/O ... 195
Bytestrings ... 198
Strict and Lazy Bytestrings ... 199
Copying Files with Bytestrings ... 201

**10**
**FUNCTIONALLY SOLVING PROBLEMS 203**

Reverse Polish Notation Calculator ... 203
Calculating RPN Expressions ... 204
Writing an RPN Function ... 205
Adding More Operators ... 207
Heathrow to London ... 208
Calculating the Quickest Path ... 209
Representing the Road System in Haskell ... 211
Writing the Optimal Path Function ... 212
Getting a Road System from the Input ... 215

**11**
**APPLICATIVE FUNCTORS 217**

Functors Redux ... 218
I/O Actions As Functors ... 218
Functions As Functors ... 220
Functor Laws ... 223
Law 1 ... 223
Law 2 ... 224
Breaking the Law ... 225
Using Applicative Functors ... 227
Say Hello to Applicative ... 228
Maybe the Applicative Functor ... 229
The Applicative Style ... 230

Lists ..... 232
IO Is An Applicative Functor, Too ..... 234
Functions As Applicatives ..... 235
Zip Lists ..... 237
Applicative Laws ..... 238
Useful Functions for Applicatives ..... 238

**12**
**MONOIDS** **243**

Wrapping an Existing Type into a New Type ..... 243
Using newtype to Make Type Class Instances ..... 246
On newtype Laziness ..... 247
type vs. newtype vs. data ..... 249
About Those Monoids ..... 250
The Monoid Type Class ..... 252
The Monoid Laws ..... 253
Meet Some Monoids ..... 253
Lists Are Monoids ..... 253
Product and Sum ..... 254
Any and All ..... 256
The Ordering Monoid ..... 257
Maybe the Monoid ..... 260
Folding with Monoids ..... 262

**13**
**A FISTFUL OF MONADS** **267**

Upgrading Our Applicative Functors ..... 267
Getting Your Feet Wet with Maybe ..... 269
The Monad Type Class ..... 272
Walk the Line ..... 274
Code, Code, Code ..... 274
I'll Fly Away ..... 276
Banana on a Wire ..... 278
do Notation ..... 280
Do As I Do ..... 282
Pierre Returns ..... 282
Pattern Matching and Failure ..... 284
The List Monad ..... 285
do Notation and List Comprehensions ..... 288
MonadPlus and the guard Function ..... 288
A Knight's Quest ..... 290
Monad Laws ..... 292
Left Identity ..... 293
Right Identity ..... 294
Associativity ..... 294

**14**
**FOR A FEW MONADS MORE** **297**

Writer? I Hardly Knew Her! ..... 298
Monoids to the Rescue ..... 300
The Writer Type ..... 302
Using do Notation with Writer ..... 303
Adding Logging to Programs ..... 304
Inefficient List Construction ..... 306
Using Difference Lists ..... 307
Comparing Performance ..... 309
Reader? Ugh, Not This Joke Again ..... 310
Functions As Monads ..... 311
The Reader Monad ..... 312
Tasteful Stateful Computations ..... 313
Stateful Computations ..... 314
Stacks and Stones ..... 314
The State Monad ..... 316
Getting and Setting State ..... 318
Randomness and the State Monad ..... 320
Error Error on the Wall ..... 321
Some Useful Monadic Functions ..... 323
liftM and Friends ..... 323
The join Function ..... 326
filterM ..... 328
foldM ..... 331
Making a Safe RPN Calculator ..... 332
Composing Monadic Functions ..... 335
Making Monads ..... 336

**15**
**ZIPPERS** **343**

Taking a Walk ..... 344
A Trail of Breadcrumbs ..... 346
Going Back Up ..... 348
Manipulating Trees Under Focus ..... 350
Going Straight to the Top, Where the Air Is Fresh and Clean! ..... 351
Focusing on Lists ..... 352
A Very Simple Filesystem ..... 353
Making a Zipper for Our Filesystem ..... 355
Manipulating a Filesystem ..... 357
Watch Your Step ..... 358
Thanks for Reading! ..... 360

**INDEX** **363**

# INTRODUCTION

## 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.

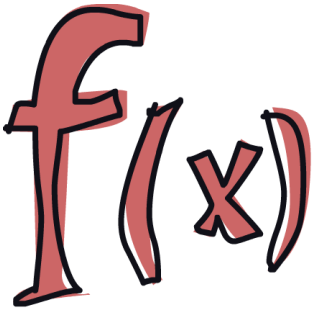
My first reaction to Haskell was that the language was just too weird. But after getting over that initial hurdle, it was smooth sailing. Even if Haskell seems strange to you at first, don't give up. Learning Haskell is almost like learning to program for the first time all over again. It's fun, and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

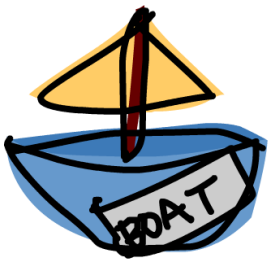
Let's look at an example of Haskell's laziness. Say you have a list of numbers, `xs = [1,2,3,4,5,6,7,8]`, and a function called `doubleMe` that doubles every element and returns the result as a new list. If you want to multiply your list by 8, your code might look something like this:

```
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.