Real World Haskell

Bryan O'Sullivan, John Goerzen, and Don Stewart

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Taipei • Tokyo

Preface

Have We Got a Deal for You!

Haskell is a deep language; we think learning it is a hugely rewarding experience. We will focus on three elements as we explain why. The first is *novelty*: we invite you to think about programming from a different and valuable perspective. The second is *power*: we'll show you how to create software that is short, fast, and safe. Lastly, we offer you a lot of *enjoyment*: the pleasure of applying beautiful programming techniques to solve real problems.

Novelty

Haskell is most likely quite different from any language you've ever used before. Compared to the usual set of concepts in a programmer's mental toolbox, functional programming offers us a profoundly different way to think about software.

In Haskell, we deemphasize code that modifies data. Instead, we focus on functions that take immutable values as input and produce new values as output. Given the same inputs, these functions always return the same results. This is a core idea behind functional programming.

Along with not modifying data, our Haskell functions usually don't talk to the external world; we call these functions *pure*. We make a strong distinction between pure code and the parts of our programs that read or write files, communicate over network connections, or make robot arms move. This makes it easier to organize, reason about, and test our programs.

We abandon some ideas that might seem fundamental, such as having a `for` loop built into the language. We have other, more flexible, ways to perform repetitive tasks.

Even the way in which we evaluate expressions is different in Haskell. We defer every computation until its result is actually needed—Haskell is a *lazy* language. Laziness is not merely a matter of moving work around, it profoundly affects how we write programs.

Power

Throughout this book, we will show you how Haskell's alternatives to the features of traditional languages are powerful and flexible and lead to reliable code. Haskell is positively crammed full of cutting-edge ideas about how to create great software.

Since pure code has no dealings with the outside world, and the data it works with is never modified, the kind of nasty surprise in which one piece of code invisibly corrupts data used by another is very rare. Whatever context we use a pure function in, the function will behave consistently.

Pure code is easier to test than code that deals with the outside world. When a function responds only to its visible inputs, we can easily state properties of its behavior that should always be true. We can automatically test that those properties hold for a huge body of random inputs, and when our tests pass, we move on. We still use traditional techniques to test code that must interact with files, networks, or exotic hardware. Since there is much less of this impure code than we would find in a traditional language, we gain much more assurance that our software is solid.

Lazy evaluation has some spooky effects. Let's say we want to find the *k* least-valued elements of an unsorted list. In a traditional language, the obvious approach would be to sort the list and take the first *k* elements, but this is expensive. For efficiency, we would instead write a special function that takes these values in one pass, and that would have to perform some moderately complex bookkeeping. In Haskell, the sort-then-take approach actually performs well: laziness ensures that the list will only be sorted enough to find the *k* minimal elements.

Better yet, our Haskell code that operates so efficiently is tiny and uses standard library functions:

```
-- file: ch00/KMinima.hs
-- lines beginning with "--" are comments.

minima k xs = take k (sort xs)
```

It can take a while to develop an intuitive feel for when lazy evaluation is important, but when we exploit it, the resulting code is often clean, brief, and efficient.

As the preceding example shows, an important aspect of Haskell's power lies in the compactness of the code we write. Compared to working in popular traditional languages, when we develop in Haskell we often write much less code, in substantially less time and with fewer bugs.

Enjoyment

We believe that it is easy to pick up the basics of Haskell programming and that you will be able to successfully write small programs within a matter of hours or days.

Since effective programming in Haskell differs greatly from other languages, you should expect that mastering both the language itself and functional programming techniques will require plenty of thought and practice.

Harking back to our own days of getting started with Haskell, the good news is that the fun begins early: it's simply an entertaining challenge to dig into a new language—in which so many commonplace ideas are different or missing—and to figure out how to write simple programs.

For us, the initial pleasure lasted as our experience grew and our understanding deepened. In other languages, it's difficult to see any connection between science and the nuts-and-bolts of programming. In Haskell, we have imported some ideas from abstract mathematics and put them to work. Even better, we find that not only are these ideas easy to pick up, but they also have a practical payoff in helping us to write more compact, reusable code.

Furthermore, we won't be putting any "brick walls" in your way. There are no especially difficult or gruesome techniques in this book that you must master in order to be able to program effectively.

That being said, Haskell is a rigorous language: it will make you perform more of your thinking up front. It can take a little while to adjust to debugging much of your code before you ever run it, in response to the compiler telling you that something about your program does not make sense. Even with years of experience, we remain astonished and pleased by how often our Haskell programs simply work on the first try, once we fix those compilation errors.

What to Expect from This Book

We started this project because a growing number of people are using Haskell to solve everyday problems. Because Haskell has its roots in academia, few of the Haskell books that currently exist focus on the problems and techniques of the typical programming that we're interested in.

With this book, we want to show you how to use functional programming and Haskell to solve realistic problems. We take a hands-on approach: every chapter contains dozens of code samples, and many contain complete applications. Here are a few examples of the libraries, techniques, and tools that we'll show you how to develop:

- Create an application that downloads podcast episodes from the Internet and stores its history in an SQL database.
- Test your code in an intuitive and powerful way. Describe properties that ought to be true, and then let the QuickCheck library generate test cases automatically.
- Take a grainy phone camera snapshot of a barcode and turn it into an identifier that you can use to query a library or bookseller's website.
- Write code that thrives on the Web. Exchange data with servers and clients written in other languages using JSON notation. Develop a concurrent link checker.

A Little Bit About You

What will you need to know before reading this book? We expect that you already know how to program, but if you've never used a functional language, that's fine.

No matter what your level of experience is, we tried to anticipate your needs; we go out of our way to explain new and potentially tricky ideas in depth, usually with examples and images to drive our points home.

As a new Haskell programmer, you'll inevitably start out writing quite a bit of code by hand for which you could have used a library function or programming technique, had you just known of its existence. We packed this book with information to help you get up to speed as quickly as possible.

Of course, there will always be a few bumps along the road. If you start out anticipating an occasional surprise or difficulty along with the fun stuff, you will have the best experience. Any rough patches you might hit won't last long.

As you become a more seasoned Haskell programmer, the way that you write code will change. Indeed, over the course of this book, the way that we present code will evolve, as we move from the basics of the language to increasingly powerful and productive features and techniques.

What to Expect from Haskell

Haskell is a general-purpose programming language. It was designed without any application niche in mind. Although it takes a strong stand on how programs should be written, it does not favor one problem domain over others.

While at its core, the language encourages a pure, lazy style of functional programming, this is the *default*, not the only option. Haskell also supports the more traditional models of procedural code and strict evaluation. Additionally, although the focus of the language is squarely on writing statically typed programs, it is possible (though rarely seen) to write Haskell code in a dynamically typed manner.