Introduction

Welcome to a new way to learn Haskell. Perhaps you are coming to this book frustrated by previous attempts to learn Haskell. Perhaps you have only the faintest notion of what Haskell is. Perhaps you are coming here because you are not convinced that anything will ever be better than Common Lisp/Scala/Ruby/whatever language you love, and you want to argue with us. Perhaps you were just looking for the 18 billionth *(n.b.: this number may be inaccurate)* monad tutorial, certain that this time around you will understand monads once and for all. Whatever your situation, welcome and read on! It is our goal here to make Haskell as clear, painless, and practical as we can, no matter what prior experiences you're bringing to the table.

Why This Book

If you are new to programming entirely, Haskell is a great first language. Haskell is a general purpose, functional programming[1] language. It's applicable virtually anywhere one would use a program to solve a problem, save for some specific embedded applications. If you could write software to solve a

[1] Functional programming is a style of programming in which function calls, rather than a series of instructions for the computer to execute, are the primary constructs of your program. What it is doesn't matter much right now; Haskell completely embodies the functional style, so it will become clear over the course of the book.

problem, you could probably use Haskell.

If you are already a programmer, you may be looking to enrich your skills by learning Haskell for a variety of reasons — from love of pure functional programming itself to wanting to write functional Scala code to finding a bridge to PureScript or Idris. Languages such as Java are gradually adopting functional concepts, but most were not designed to be functional languages. Because Haskell is a pure functional language, it is a fertile environment for mastering functional programming. That way of thinking and problem solving is useful, no matter what other languages you might know or learn. We've heard from readers who are finding this book useful to their work in diverse languages such as Scala, F#, Frege, Swift, PureScript, Idris, and Elm.

Haskell has a bit of a reputation for being difficult. Writing Haskell may seem to be more difficult up front, not just because of the hassle of learning a language that is syntactically and conceptually different from a language you already know, but also because of features such as strong typing that enforce some discipline in how you write your code. But what seems like a bug is a feature. Humans, unfortunately, have relatively limited abilities of short-term memory and concentration, even if we don't like to admit it. We *cannot* track all relevant metadata about our programs in our heads. Using up working memory for anything a computer can do for us is counterproductive, and computers are very good at keeping track of

data for us, including metadata such as types.

We don't write Haskell because we're geniuses — we use tools like Haskell because they help us. Good tools like Haskell enable us to work faster, make fewer mistakes, and have more information about what our code is supposed to do as we read it.

We use Haskell because it is easier (over the long run) and enables us to do a better job. That's it. There's a ramp-up required in order to get started, but that can be ameliorated with patience and a willingness to work through exercises.

OK, but I was just looking for a monad tutorial...

The bad news is looking for an easy route into Haskell and functional programming is how a lot of people end up thinking it's "too hard" for them. The good news is we have a lot of experience teaching and we don't want that to happen to anyone, but especially not you, gentle reader.

We encourage you to forget what you might already know about programming and come at this course in Haskell with a beginner's mindset. Make yourself an empty vessel, ready to let the types flow through you.

If you are an experienced programmer, learning Haskell is more like learning to program all over again. Learning Haskell imposes new ways of thinking about and structuring programs on most people already comfortable with an imperative or

untyped programming language. This makes it harder to learn not because it is intrinsically harder, but because most people who have learned at least a couple of programming languages are accustomed to the process being trivial, and their expectations have been set in a way that lends itself to burnout and failure.

If Haskell is your first language, or even if it is not, you may have noticed a specific problem with many Haskell learning resources: they assume a certain level of background with programming, so they frequently explain Haskell concepts in terms, by analogy or by contrast, of programming concepts from other languages. This is confusing for the student who doesn't know those other languages, but we posit that it is just as unhelpful for experienced programmers. Most attempts to compare Haskell with other languages only lead to a superficial understanding of Haskell, and making analogies to loops and other such constructs can lead to bad intuitions about how Haskell code works. For all of these reasons, we have tried to avoid relying on knowledge of other programming languages. Just as you can't achieve fluency in a human language so long as you are still attempting direct translations of concepts and structures from your native language to the target language, it's best to learn to understand Haskell on its own terms.

But I've heard Haskell is hard...

There's a wild rumor that goes around the internet from time to time about needing a Ph.D. in mathematics and an understanding of monads just to write "hello, world"[2] in Haskell.

We will write "hello, world" in Chapter 3. We're going to do some arithmetic before that to get you used to function syntax and application in Haskell, but you will not need a Ph.D. in monadology to write it.

In truth, there will be a monad underlying our "hello, world," and by the end of the book, you *will* understand monads, but you'll be interacting with monadic code long before you understand how it all works. You'll find, at times, this book goes into more detail than you strictly need to be able to write Haskell successfully. There is no problem with that. You do not need to understand everything in here perfectly on the first try.

You are not a Spartan warrior who must come back with your shield or on it. Returning later to investigate things more deeply is an efficient technique, not a failure.

[2]Writing "hello, world" in a new programming language is a standard sort of "baby's first program," so the idea here is that if it's difficult to write a "hello, world" program, then the language must be impossible. There are languages that have purposely made it inhumanly difficult to write such programs, but Haskell is not one of them.

A few words to new programmers

We've tried very hard to make this book as accessible as possible, no matter your level of previous experience. We have kept comparisons and mentions of other languages to a minimum, and we promise that if we compare something in Haskell to something in another language, that comparison is not itself crucial to understanding the Haskell — it's just a little extra for those who do know the other language.

However, especially as the book progresses and the exercises and projects get more "real," there are going to be terms and concepts that we do not have the space to explain fully but that are relatively well known among programmers. You may have to do internet searches for terms like *JSON*. The next section of this introduction references things that you may not know about but programmers will — don't panic. We think you'll still get something out of reading it, but if not, it's not something to worry about. The fact that you don't know every term in this book before you come to it is not a sign that you can't learn Haskell or aren't ready for this: it's only a sign that you don't *know everything yet*, and since no one does, you're in fine company.

Along those same lines, this book does not offer much instruction on using the terminal and text editor. The instructions provided assume you know how to find your way around your terminal and understand how to do simple tasks like

make a directory or open a file. Due to the number of text editors available, we do not provide specific instructions for any of them.[3]

If you need help or would like to start getting to know the communities of functional programmers, there are several options. The Freenode IRC channel `#haskell-beginners` has teachers who will be glad to help you, and they especially welcome questions regarding specific problems that you are trying to solve.[4] There are also Slack channels and subreddits where Haskellers congregate, along with a plethora of Haskell-oriented blogs, many of which are mentioned in footnotes and recommended readings throughout the book. Many of our readers also program in languages like Swift and Scala, so you may want to investigate those communities as well.

Haskevangelism

The rest of this introduction will give some background of Haskell and will make reference to other programming lan-

[3] If you're quite new and unsure what to do about text editors, you might consider Atom. It's free, open-source, and configurable. Sublime Text has served Julie well throughout the writing of the book, but is not free. Chris uses Emacs most of the time; Emacs is very popular among programmers, but has its own learning curve. Vim is another popular text editor with *its* own learning curve. If you have no experience with Emacs or Vim, we'd really recommend sticking with something like Sublime or Atom for now.

[4] Freenode IRC (Internet Relay Chat) is a network of channels for textual chat. There are other IRC networks around, as well as other group chat platforms, but the Freenode IRC channels for Haskell are popular meeting places for the Haskell community. There are several ways to access Freenode IRC, including Irssi and HexChat, if you're interested in getting to know the community in their natural habitat.

guages and styles. If you're a new programmer, it is possible not all of this will make sense, and that's okay. The rest of the book is written with beginners in mind, and the features we're outlining will make more sense as you work through the book.

We're going to compare Haskell a bit with other languages to demonstrate why we think using Haskell is valuable. Haskell is a language in a progression of languages dating back to 1973, when ML was invented by Robin Milner and others at the University of Edinburgh. ML was itself influenced by ISWIM, which was in turn influenced by ALGOL 60 and Lisp. We mention this lineage because Haskell *isn't* new. The most popular implementation of Haskell, the Glasgow Haskell Compiler (GHC), is mature and well-made. Haskell brings together some nice design choices that make for a language that offers more expressiveness than Ruby, but more type safety than any language presently in wide use commercially.

In 1968, the ALGOL68 dialect had the following features built into the language:

1. User-defined record types.
2. User-defined sum types (unions not limited to simple enumerations).
3. Switch/case expressions supporting the sum types.
4. Compile-time enforced constant values, declared with `=` rather than `:=`.

5. Unified syntax for using value and reference types — no manual pointer dereferencing.
6. Closures with lexical scoping (without this, many functional patterns fall apart).
7. Implementation-agnostic parallelized execution of procedures.
8. Multi-pass compilation — you can declare stuff after you use it.

As of the early 21st century, many popular languages used commercially don't have anything equivalent to or better than what ALGOL68 had. We mention this because we believe technological progress in computer science, programming, and programming languages is possible, desirable, and critical to software becoming a true engineering discipline. By that, we mean that while the phrase "software engineering" is in common use, engineering disciplines involve the application of both scientific and practical knowledge to the creation and maintenance of better systems. As the available materials change and as knowledge grows, so must engineers.

Haskell leverages more of the developments in programming languages invented since ALGOL68 than most languages in popular use, but with the added benefit of a mature implementation and sound design. Sometimes we hear Haskell being dismissed as "academic" because it is relatively up-to-date

with the current state of mathematics and computer science research. In our view, that progress is good and helps us solve practical problems in modern computing and software design.

Progress is possible and desirable, but it is not monotonic or inevitable. The history of the world is riddled with examples of uneven progress. For example, it is estimated that scurvy killed two million sailors between the years 1500 and 1800. Western culture has forgotten the cure for scurvy multiple times. As early as 1614, the Surgeon General of the East India Company recommended bringing citrus on voyages for scurvy. It saved lives, but the understanding of *why* citrus cured scurvy was incorrect. This led to the use of limes, which have a lower vitamin C content than lemons, and scurvy returned until ascorbic acid was discovered in 1932. Indiscipline and stubbornness (the British Navy stuck with limes despite sailors continuing to die from scurvy) can hold back progress. We'd rather have a doctor who is willing to understand that he makes mistakes, will be responsive to new information, and even actively seek to expand his understanding rather than one that hunkers down with a pet theory informed by anecdote.

There are other ways to prevent scurvy, just as there are other programming languages you can use to write software. Or perhaps you are an explorer who doesn't believe scurvy can happen to you. But packing lemons provides some insurance on those long voyages. Similarly, having Haskell in

your toolkit, even when it's not your only tool, provides type safety and predictability that can improve your software development. Buggy software might not literally make your teeth fall out, but software problems are far from trivial, and when there are better ways to solve those problems — not perfect, but better — it's worth your time to investigate them.

Set your limes aside for now, and join us at the lemonade stand.

What's in this book?

This book is more of a course than a book, something to be worked through. There are exercises sprinkled liberally throughout the book; we encourage you to do them, even when they seem simple. Those exercises are where the majority of your epiphanies will come from. No amount of chattering, no matter how well structured and suited to your temperament, will be as effective as *doing the work*. If you do get to a later chapter and find you did not understand a concept or structure well enough, you may want to return to an earlier chapter and do more exercises until you understand it.

We believe that spaced repetition and iterative deepening are effective strategies for learning, and the structure of the book reflects this. You may notice we mention something only briefly at first, then return to it over and over. As your experience with Haskell deepens, you have a base from which

to move to a deeper level of understanding. Try not to worry that you don't understand something completely the first time we mention it. By moving through the exercises and returning to concepts, you can develop a solid intuition for functional programming.

The exercises in the first few chapters are designed to rapidly familiarize you with basic Haskell syntax and type signatures, but you should expect exercises to grow more challenging in each successive chapter. Where possible, reason through the code samples and exercises in your head first, then type them out — either into the REPL[5] or into a source file — and check to see if you were right. This will maximize your ability to understand and reason about programs and about Haskell. Later exercises may be difficult. If you get stuck on an exercise for an extended period of time, proceed and return to it at a later date.

We cover a mix of practical and abstract matters required to use Haskell for a wide variety of projects. Chris's experience is principally with production backend systems and frontend web applications. Julie is a linguist and teacher by training and education, and learning Haskell was her first experience with computer programming. The educational priorities of this book are biased by those experiences. Our goal is to help

[5] This is short for read-eval-print loop, an interactive programming shell that evaluates expressions and returns results in the same environment. The REPL we'll be using is called GHCi — 'i' for "interactive."

you not just write typesafe functional code but to understand it on a deep enough level that you can go from here to more advanced Haskell projects in a variety of ways, depending on your own interests and priorities.

Each chapter focuses on different aspects of a particular topic. We start with a short introduction to the lambda calculus. What does this have to do with programming? All modern functional languages are based on the lambda calculus, and a passing familiarity with it will help you down the road with Haskell. If you've understood the lambda calculus, understanding the feature known as *currying* will be a breeze, for example.

The next few chapters cover basic expressions and functions in Haskell, some simple operations with strings (text), and a few essential types. You may feel a strong temptation, especially if you have programmed previously, to skim or skip those first chapters. *Please do not do this.* Even if those first chapters are covering concepts you're familiar with, it's important to spend time getting comfortable with Haskell's terse syntax, making sure you understand the difference between working in the REPL and working in source files, and becoming familiar with the compiler's sometimes quirky error messages. Certainly you may work quickly through those chapters — just don't skip them.

From there, we build both outward and upward so that your understanding of Haskell both broadens and deepens. When

you finish this book, you will not just know what monads are, you will know how to use them effectively in your own programs and understand the underlying algebra involved. We promise — you will. We only ask that you do not go on to write a monad tutorial on your blog that explains how monads are really just like jalapeno poppers.

In each chapter you can expect:

- additions to your vocabulary of standard functions;
- syntactic patterns that build on each other;
- theoretical foundations so you understand how Haskell works;
- illustrative examples of how to read Haskell code;
- step-by-step demonstrations of how to write your own functions;
- explanations of how to read common error messages and how to avoid those errors;
- exercises of varying difficulty sprinkled throughout;
- definitions of important terms.

We have put definitions at the end of most chapters. Each term is, of course, defined within the body of the chapter, but

we added separate definitions at the end as a point of review. If you've taken some time off between one chapter and the next, the definitions can remind you of what you have already learned, and, of course, they may be referred to any time you need a refresher.

There are also recommendations at the end of most chapters for followup reading. They are certainly not required but are resources we personally found accessible and helpful that may help you learn more about topics covered in the chapter.

Best practices for examples and exercises

We have tried to include a variety of examples and exercises in each chapter. While we have made every effort to include only exercises that serve a clear pedagogical purpose, we recognize that not all individuals enjoy or learn as much from every type of demonstration or exercise. Also, since our readers will necessarily come to the book with different backgrounds, some exercises may seem too easy or difficult to you but be just right for someone else. Do your best to work through as many exercises as seems practical for you. But if you skip all the types and typeclasses exercises and then find yourself confused when we get to Monoid, by all means, come back and do more exercises until you understand.

Here are a few things to keep in mind to get the most out of them:

- Examples are usually designed to demonstrate, with real code, what we've just talked or are about to talk about in further detail.

- You are intended to *type* all of the examples into the REPL or a file and load them. We *strongly* encourage you to attempt to modify the example and play with the code after you've made it work. Forming hypotheses about what effect changes will have and verifying them is critical! It is better to type the code examples and exercises yourself rather than copy and paste because typing makes you pay more attention to it.

- Sometimes the examples are designed intentionally to be broken. Check surrounding prose if you're confused by an unexpected error as we will not show you code that doesn't work without commenting on the breakage. If it's still broken and it's not supposed to be, you should start checking your syntax and formatting for errors.

- Not every example is designed to be entered into the REPL; not every example is designed to be entered into a file. Once we have explained the syntactic differences between files and REPL expressions, you are expected to perform the translation between the two yourself. You should be accustomed to working with code in an interactive manner by the time you finish the book. You'll want

to gradually move away from typing code examples and exercises, except in limited cases, directly into GHCi and develop the habit of working in source files. Editing and modifying code, as you will be doing a lot as you rework exercises, is easier and more practical in a source file. You will still load your code into GHCi to run it.

- You may want to keep exercises, especially longer ones, as named modules. There are several exercises, especially later in the book, that we return to several times and being able to reload the work you've already done and add only the new parts will save you a lot of time and grief. We have tried to note some of the exercises where this will be especially helpful.

- Exercises at the end of the chapter may include some review questions covering material from previous chapters and are more or less ordered from least to most challenging. Your mileage may vary.

- Even exercises that seem easy can increase your fluency in a topic. We do not fetishize difficulty for difficulty's sake. We just want you to understand the topics as well as possible. That can mean coming at the same problem from different angles.

- We ask you to write and then rewrite (using different syntax) a lot of functions. Few problems have only one

possible solution, and solving the same problem in different ways increases your fluency and comfort with the way Haskell works (its syntax, its semantics, and in some cases, its evaluation order).

- Do not feel obligated to do all the exercises in a single sitting or even in a first pass through the chapter. In fact, spaced repetition is generally a more effective strategy.

- Some exercises, particularly in the earlier chapters, may seem very contrived. Well, they are. But they are contrived to pinpoint certain lessons. As the book goes on and you have more Haskell under your belt, the exercises become less contrived and more like "real Haskell."

- Another benefit to writing code in a source file and then loading it into the REPL is that you can write comments about the process you went through in solving a problem. Writing out your own thought process can clarify your thoughts and make the solving of similar problems easier. At the very least, you can refer back to your comments and learn from yourself.

- Sometimes we intentionally underspecify function definitions. You'll commonly see things like:

```
f = undefined
```

Even when f will probably take named arguments in your implementation, we're not going to name them for you. Nobody will scaffold your code for you in your future projects, so don't expect this book to either.