Haskell Programming

from first principles

Christopher Allen
Julie Moronuki

Pure functional programming

without fear or frustration

Reader feedback

Astonishingly insightful examples. This book is a lot like having a good teacher—it never fails to provide the low-end information even though I have already moved on. So just like a good teacher isn't presumptuous in what I'm supposed to know (which might force me to try and save face in case I do not, yet), information conveniently resurfaces.

—David Deutsch

When @haskellbook is done, it will be an unexpected milestone for #haskell. There will forever be Haskell before, and Haskell after.

—Jason Kuhrt

I feel safe recommending Haskell to beginners now that @haskellbook is available, which is very beginner friendly.

—Gabriel Gonzalez

"Structure and Interpretation of Computer Programs" has its credit, but @haskellbook is now my #1 recommendation for FP beginners.

—Irio Musskopf

The book is long, but not slow—a large fraction of it is made up of examples and exercises. You can tell it's written by someone who's taught Haskell to programmers before.

—Christopher Jones

I already have a lot of experience with Haskell, but I've never felt confident in it the way this book has made me feel.

—Alain O'Dea

Real deal with @haskellbook is that you don't just learn Haskell; you get a hands on experience as to why functional programming works.

—George Makrydakis

One of my goals this year is to evangelize @haskellbook and @Haskell-ForMac. I think these tools will make anyone who uses them better. I want to get comfortable with it so that I can shift how I think about Swift.

—Janie Clayton

Contents

Preface

Chris's story

I've been programming for over 18 years, 11 of them professionally. I've worked primarily in Common Lisp, Clojure, and Python. I became interested in Haskell about 6 years ago. Haskell was the language that made me aware that progress is being made in programming language research and that there are benefits to using a language with a design informed by knowledge of those advancements.

I've had type errors in Clojure that multiple professional Clojure developers (including myself) couldn't resolve in fewer than two hours because of the source-to-sink distance caused by dynamic typing. We had copious tests. We added `println`s everywhere. We tested individual functions from the REPL. It still took ages. It was only 250 lines of Clojure. I did finally fix it and found it was due to vectors in Clojure implementing `IFn`. The crazy values that propagated from the `IFn` usage of the vector allowed malformed data to propagate downward far away from the origin of the problem. I've had similar things happen in Python and Common Lisp, as well. The same issue in Haskell would be resolved trivially, in a minute or less, because the type checker can identify precisely where you are inconsistent.

I use Haskell, because I want to be able to refactor without fear, because I want maintenance to be something I don't resent, so I can reuse code freely. This doesn't come without learning new things. The difference between people who are "good at math" — who "do it in their head" — and professional mathematicians is that the latter show their work and use tools that help them get the job done. When you're using a dynamically typed language, you're forcing yourself unnecessarily to do it "in your head." As a human with limited working memory, I want all the help I can get to reason about and write correct code. Haskell provides that help.

Haskell is not a difficult language to use—quite the opposite. I'm now able to tackle problems that I couldn't have tackled when I was primarily a Clojure, Common Lisp, or Python user. Haskell is difficult to teach effectively, and the ineffective didactic method has made it hard for many people to learn.

It doesn't have to be that way.

I spent two years actively teaching Haskell online and in person

before starting on this book. Along the way, I started keeping notes on exercises and methods of teaching specific concepts and techniques that worked. Those notes eventually turned into my guide for learning Haskell. I'm still learning how to teach Haskell better by working with people locally in Austin, Texas, as well as online in the IRC channel I made for beginners to get help with learning Haskell.

I wrote this book, because I had a hard time learning Haskell, and I don't want others to struggle the way I did.

Acknowledgements

This book developed out of many efforts to teach and learn Haskell, online and off. We could not have done this without the help of the growing community of friendly Haskellers as well as the Haskell learners who have graciously offered time to help us make the book better.

First and foremost, we owe a huge debt of gratitude to our first-round reviewers, Angela May O'Connor and Martin Vlk, for their tremendous patience. We sent them both some very rough material, and they have been willing to work with it and send detailed feedback about what worked and what didn't. Their reviews helped ensure the book is both suitable for beginners and comprehensive. Also, they're both just wonderful people all around.

Martin DeMello, Daniel Gee, and Simon Yang have each sent us (many) smart criticisms and helpful suggestions. The book would have been shorter without their help, we think, but it's much more thorough and clear now.

A number of people have contributed feedback and technical review for limited parts of the book. Thanks to Sean Chalmers, Erik de Castro Lopo, Alp Mestanogullari, Juan Alberto Sanchez, Jonathan Ferguson, Deborah Newton, Matt Parsons, Peter Harpending, Josh Cartwright, Eric Mertens, and George Makrydakis, who have all offered critiques of our writing and our technical coverage of different topics.

We have some very active early access readers who send us a stream of feedback, everything from minor typographical errors they find to questions about exercises, and we're pleased and grateful to have had their input. The book would be messier and the exercises less useful if not for their help. Julien Baley and Jason Kuhrt have been particularly outstanding on this front, not only representing a nontrivial portion of our reader feedback over the course of several releases of the book but also catching things nobody else noticed.

The book cover was designed by David Deutsch (@skore_de on Twitter). He took pity on the state of our previous, original, super-special early-access cover and took it upon himself to redesign it. We liked it so much we asked him to redo the book's website, as well. He's a talented designer, and we're grateful for all the work he's done

for us.

A special thank-you is owed to Soryu Moronuki, Julie's son, who agreed to try to use the book to teach himself Haskell and allowed us to use his feedback and occasionally blog about his progress.

A warm hello to all the reading groups, both online and in meatspace, that have formed to work through the book together. We've had some great feedback from these groups and hope to visit with you all someday. We're delighted to see the Haskell community growing.

We would also like to thank Michael Neale for being funny and letting us use something he said on Twitter as an epigraph. Some day we hope to buy the gentleman a beer.

Thank you as well to Steven Proctor, for having hosted us on his Functional Geekery podcast, and to Adam Stacoviak and Jerod Santo for inviting us onto their podcast, The Changelog—and to Zaki Manian for bringing us to Adam and Jerod's attention.

This book's final release would not have been possible without Steven Syrek's patient and gracious help. Not only did he read the book as a learner, but I learned a great deal about how to run a reading group from him. On top of all that, Steve completed the book's final edit and proofcheck in his very limited spare time. Steve's meticulous eye for poor grammar, unclear language, and typographical errors has made this book ready to publish.

Any errors in the book remain the sole responsibility of the authors.

Chris I would like to thank the participants in the `#haskell-beginners` IRC channel, the teachers and the students, who have helped me practice and refine my teaching techniques. Many of the exercises and approaches in the book would never have happened without the wonderful Haskell IRC community to learn from.

I owe Alex Kurilin, Carter Schonwald, Aidan Coyne, and Mark Wotton thanks for being there when I was *really* bad at teaching, being kind and patient friends, and for giving me advice when I needed it. I wouldn't have scratched this itch without y'all.

Julie I would like to send a special shout-out to the Austin Haskell Meetup group, especially Sukant Hajra and Austin Seipp for giving me the opportunity to teach at the Meetup.

The list of Haskellers who have responded to the kvetches and confusions of a Haskell beginner with assistance, humor, and advice would be very long indeed, but I owe special gratitude to Sooraj Bhat, Reid McKenzie, Dan Lien, Zaki Manian, and Alex Feldman-Crough for their help and encouragement. I wouldn't have made it through the last few months of finishing this thing without the patient advice and friendship of Chris Martin.

My husband and children have tolerated me spending uncountable hours immersed in the dark arts of thunkery. I am grateful for their love, patience, and support and hope that my kids will remember this: that it's never too late to learn something new. *Besos, mijos.*

Finally, a warm thank you to George Makrydakis for the ongoing discussion on matters to do with math, programming, and the weirding way.

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 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.

[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.

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 counter-productive, 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 that 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. Become 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, by analogy or by contrast, in terms 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, that 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.

[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.

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.

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 a 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

[3]If you're quite new and unsure what to do about text editors, you might consider Visual Studio Code. It's free, open-source, and configurable. Sublime Text has served Julie well throughout the writing of the book, but it 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 VS Code or Sublime Text for now.

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 provide some background on Haskell and make reference to other programming languages and styles. If you're a new programmer, it is possible that not all of this will make sense, and that's OK. 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 one 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 also 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 that were not limited to simple enumerations).

[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.

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. And yet, 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 who 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 the 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 you not just write type-safe 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 understand 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

[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."

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 jalapeño 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 follow-up reading. They are certainly not required but are resources we personally found accessible and helpful, and they may help you learn more about certain topics covered in their respective 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 didactic 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 exercises on types and type classes 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 them.
- Sometimes, the examples are designed intentionally to be broken. Check the 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.

- 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.
- 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 a 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 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."

- Sometimes, we intentionally under-specify 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.

Chapter 1

All You Need is Lambda

> Even the greatest mathematicians, the ones that we would put into our mythology of great mathematicians, had to do a great deal of leg work in order to get to the solution in the end.
>
> Daniel Tammett

1.1 All You Need is Lambda

This chapter provides a very brief introduction to the lambda calculus, a model of computation devised in the 1930s by Alonzo Church. A calculus is a method of calculation or reasoning; the lambda calculus is one process for formalizing a method. Like Turing machines, the lambda calculus formalizes the concept of effective computability, thus determining which problems, or classes of problems, can be solved.

You may be wondering where the Haskell is. You may be contemplating skipping this chapter. You may feel tempted to leap ahead to the fun stuff where we build a project.

DON'T.

We're starting from first principles here, so that when we get around to building projects, you know what you're doing. You don't start building a house from the attic down; you start from the foundation. Lambda calculus is your foundation, because Haskell is a lambda calculus.

1.2 What is functional programming?

Functional programming is a computer programming paradigm that relies on functions modeled on mathematical functions. The essence of functional programming is that programs are a combination of *expressions*. Expressions include concrete values, variables, and also functions. Functions have a more specific definition: they are expressions that are applied to an argument or input and, once applied, can be *reduced* or *evaluated*. In Haskell, and in functional programming more generally, functions are *first-class*: they can be used as values or passed as arguments, or inputs, to yet more functions. We'll define these terms more carefully as we progress through the chapter.

Functional programming languages are all based on the lambda calculus. Some languages in this general category incorporate features that aren't translatable into lambda expressions. Haskell is a *pure* functional language, because it does not. We'll further address this notion of purity later in the book, but it isn't a judgment of the moral worth of other languages.

The word *purity* in functional programming is sometimes also used to mean what is more properly called *referential transparency*. Referential transparency means that the same function, given the same values to evaluate, will always return the same result in pure functional programming, as they do in math.

Haskell's pure functional basis also lends it a high degree of abstraction and composability. Abstraction allows you to write shorter, more concise programs by factoring common, repeated structures into more generic code that can be reused. Haskell programs are built from separate, independent functions, kind of like LEGO®: the functions are bricks that can be assembled and reassembled.

These features also make Haskell's syntax rather minimalist, as you'll soon see.

1.3 What is a function?

If we step back from using the word "lambda," you most likely already know what a function is. A function is a relation between a set of possible inputs and a set of possible outputs. The function itself defines and represents that relationship. When you apply a function such as addition to two inputs, it maps those two inputs to an output—the sum of those numbers.

For example, let's imagine a function named f that defines the following relations where the first value is the input and the second is the output:

$$f(1) = A$$
$$f(2) = B$$
$$f(3) = C$$

The input set is {1, 2, 3}, and the output set is {A, B, C}.[1] A crucial point about how these relations are defined: our hypothetical function will *always* return the value A given the input 1—no exceptions!

[1]For those who would like precise terminology, the input set is known as the domain. The set of possible outputs for the function is called the codomain. Domains and codomains are sets of unique values. The subset of the codomain that contains possible outputs of the function is known as the image. The mapping between the

In contrast, the following is *not* a valid function:

$$f(1) = X$$
$$f(1) = Y$$
$$f(2) = Z$$

This gets back to the notion of referential transparency we mentioned earlier: given the same input, the output should be predictable.

Is the following function valid?

$$f(1) = A$$
$$f(2) = A$$
$$f(3) = A$$

Yes, having the same output for more than one input is valid. Imagine, for example, that you need a function that tests a positive integer for being less than 10. You'd want it to return "true" when the input is less than 10 and "false" for all other cases. In that case, several different inputs will result in the output "true"; many more will give a result of "false." Different inputs can lead to the same output.

What matters here is that the relationship of inputs and outputs is defined by the function and that the output is predictable when you know the input and the function definition.

In the above examples, we didn't demonstrate a relationship between the inputs and outputs. Let's look at an example that does define the relationship. This function is again named f:

$$f(x) = x + 1$$

This function takes one argument, which we have named x. The relationship between the input, x, and the output is described in the function body. It will add 1 to whatever value x is and return that result. When we apply this function to a value, such as 1, we substitute the value for x:

---

domain and the image or codomain need not be one-to-one; in some cases, multiple input values will map to the same value in the image, as when a function returns either "true" or "false," so that many different inputs map to each of those output values. However, a given input should not map to multiple outputs.

$$f(1) = 1 + 1$$

f applied to 1 equals 1 + 1. That tells us how to map the input to an output: 1 added to 1 becomes 2:

$$f(1) = 2$$

Understanding functions in this way—as a mapping of a set of inputs to a set of outputs—is crucial to understanding functional programming.

1.4 The structure of lambda expressions

The lambda calculus has three basic components, or *lambda terms*: expressions, variables, and abstractions. The word *expression* refers to a superset of all those things: an expression can be a variable name, an abstraction, or a combination of those things. The simplest expression is a single variable. Variables here have no meaning or value; they are only names for potential inputs to functions.

An *abstraction* is a *function*. It is a lambda term that has a head (a lambda) and a body and is applied to an argument. An *argument* is an input value.

Abstractions consist of two parts: the *head* and the *body*. The head of the function is a λ (lambda) followed by a variable name. The body of the function is another expression. So, a simple function might look like this:

$$\lambda x.x$$

The variable named in the head is the *parameter* and *binds* all instances of that same variable in the body of the function. That means, when we apply this function to an argument, each x in the body of the function will have the value of that argument. We'll demonstrate this in the next section. The act of applying a lambda function to an argument is called *application*, and application is the lynchpin of the lambda calculus.

In the previous section, we were talking about functions called f, but the lambda abstraction $\lambda x.x$ has no name. It is an *anonymous func-*

tion. A named function can be called by name by another function; an anonymous function cannot.

Let's break down the basic structure:

```
λ x . x
^─┬─^
  └──── extent of the head of the lambda.

λ x . x
  ^──── the single parameter of the
        function. This binds any
        variables with the same name
        in the body of the function.


λ x . x
      ^── body, the expression the lambda
          returns when applied. This is a
          bound variable.
```

The dot (.) separates the parameters of the lambda from the function body.

The abstraction as a whole has no name, but the reason we call it an *abstraction* is that it is a generalization, or abstraction, from a concrete instance of a problem, and it abstracts through the introduction of names. The names stand for particular values, but by using named variables, we allow for the possibility of applying the general function to different values (or, perhaps even values of different *types*, as we'll see later). When we apply the abstraction to arguments, we replace the names with values, making it concrete.

Alpha equivalence

Often, when people express this function in lambda calculus, you'll see something like this:

$$\lambda x.x$$

The variable x here is not semantically meaningful except in its role in that single expression. Because of this, there's a form of

equivalence between lambda terms called *alpha equivalence*. This is a way of saying that the following expressions all mean the same thing:

$$\lambda x.x$$
$$\lambda d.d$$
$$\lambda z.z$$

In principle, they're all the same function.

Let's look next at what happens when we apply this abstraction to a value.

1.5 Beta reduction

When we apply a function to an argument, we substitute the input expression for all instances of bound variables within the body of the abstraction. You also eliminate the head of the abstraction, since its only purpose is to bind a variable. This process is called *beta reduction*.

Let's use the function we had above:

$$\lambda x.x$$

We'll do our first beta reduction using a number.[2] We apply the function above to 2, substitute 2 for each bound variable in the body of the function, and eliminate the head:

$$(\lambda x.x)\ 2$$
$$2$$

The only bound variable is the single x, so applying this function to 2 returns 2. This function is the *identity* function.[3] All it does is accept a single argument x and return that same argument. The x has no inherent meaning, but, because it is bound in the head of this function, when the function is applied to an argument, all instances of x within the function body must have the same value.

[2]The lambda calculus can derive numbers from lambda abstractions, rather than using the numerals we are familiar with, but the applications can become quite cumbersome and difficult to read.

[3]Note that this is the same as the identity function in mathematical notation: $f(x) = x$. One difference is that $f(x) = x$ is a declaration involving a function named *f*, while the above lambda abstraction *is* a function.

Let's use an example that mixes some arithmetic into our lambda calculus. We use the parentheses here to clarify that the body expression is $x + 1$. In other words, we are not applying the function to the value 1:

$$(\lambda x.x + 1)$$

What is the result if we apply this abstraction to 2? How about to 10?

Beta reduction is this process of applying a lambda term to an argument, replacing the bound variables with the value of the argument, and eliminating the head. Eliminating the head tells you the function has been applied.

We can also apply our identity function to another lambda abstraction:

$$(\lambda x.x)(\lambda y.y)$$

In this case, we'd substitute the entire abstraction for x. We'll use a new syntax here, $[x := z]$, to indicate that z will be substituted for all occurrences of x (here z is the function $\lambda y.y$). We reduce this application like this:

$$\begin{aligned}(\lambda x.x)(\lambda y.y)\\ [x := (\lambda y.y)]\\ \lambda y.y\end{aligned}$$

Our final result is another identity function. There is no argument to apply it to, so we have nothing to reduce.

Once more, but this time we'll add another argument:

$$(\lambda x.x)(\lambda y.y)z$$

Applications in the lambda calculus are *left associative*. Unless specific parentheses suggest otherwise, they associate, or group, to the left. So, this:

$$(\lambda x.x)(\lambda y.y)z$$

Can be rewritten as:

$$((\lambda x.x)(\lambda y.y))z$$

Onward with the reduction:

$$\begin{aligned} &((\lambda x.x)(\lambda y.y))z \\ &[x := (\lambda y.y)] \\ &(\lambda y.y)z \\ &[y := z] \\ &z \end{aligned}$$

We can't reduce this any further, as there is nothing left to apply, and we know nothing about z.

We'll look at functions below that have multiple heads and also *free variables* (that is, variables in the body that are not bound by the head), but the basic process will remain the same. Beta reduction stops when there are no longer unevaluated functions applied to arguments. A computation consists of an initial lambda expression plus a finite sequence of lambda terms, each deduced from the preceding term by one application of beta reduction. We keep following the rules of application, substituting arguments for bound variables until there are no more heads left to evaluate or no more arguments to apply them to.

Free variables

The purpose of the head of the function is to tell us which variables to replace when we apply our function, that is, to bind the variables. A bound variable must have the same value throughout the expression.

But sometimes, the body expression has variables that are not named in the head. We call those variables *free variables*. In the following expression:

$$\lambda x.xy$$

The x in the body is a bound variable, because it is named in the head of the function, while the y is a free variable, because it is not. When we apply this function to an argument, nothing can be done with the y. It remains irreducible.

That whole abstraction can be applied to an argument, z, like this: $(\lambda x.xy)z$. We'll show an intermediate step, using the := syntax we introduced above, that most lambda calculus literature does not show:

1. $(\lambda x.xy)z$
   We apply the lambda to the argument z.

2. $(\lambda[x := z].xy)$
   Since x is the bound variable, all instances of x in the body of the function will be replaced with z. The head will be eliminated, and we replace any x in the body with a z.

3. zy
   The head has been applied away, and there are no more heads or bound variables. Since we know nothing about z or y, we can reduce this no further.

Note that alpha equivalence does not apply to free variables. That is, $\lambda x.xz$ and $\lambda x.xy$ are not equivalent, because z and y might be different things. However, $\lambda xy.yx$ and $\lambda ab.ba$ are equivalent due to alpha equivalence, as are $\lambda x.xz$ and $\lambda y.yz$, because the free variable is left alone.

1.6 Multiple arguments

Each lambda can only bind one parameter and can only accept one argument. Functions that require multiple arguments have multiple, nested heads. When you apply it once and eliminate the first (leftmost) head, the next one is applied and so on. This formulation was originally discovered by Moses Schönfinkel in the 1920s but was later rediscovered and named after Haskell Curry and is commonly called *currying*.

What we mean by this description is that the following:

$$\lambda xy.xy$$

Is a convenient shorthand for two nested lambdas (one for each argument, x and y):

$$\lambda x.(\lambda y.xy)$$

When you apply the first argument, you're binding x, eliminating the outer lambda, and have $\lambda y.xy$ with x being whatever the outer lambda was bound to.

To try to make this a little more concrete, let's suppose that we apply these lambdas to specific values. First, a simple example with the identity function:

1. $\lambda x.x$
2. $(\lambda x.x)\ 1$
3. $[x := 1]$
4. 1

Now let's look at a "multiple" argument lambda:

1. $\lambda xy.xy$
2. $(\lambda xy.xy)\ 1\ 2$
3. $(\lambda x.(\lambda y.xy))\ 1\ 2$
4. $[x := 1]$
5. $(\lambda y.1y)\ 2$
6. $[y := 2]$
7. $1\ 2$

That wasn't too interesting, because it's like nested identity functions! We can't meaningfully apply a 1 to a 2. Let's try something different:

1. $\lambda xy.xy$
2. $(\lambda xy.xy)(\lambda z.a)\ 1$
3. $(\lambda x.(\lambda y.xy))(\lambda z.a)\ 1$
4. $[x := (\lambda z.a)]$
5. $(\lambda y.(\lambda z.a)y)\ 1$
6. $[y := 1]$

7. $(\lambda z.a)\ 1$
   We still can apply this one more time.

8. $[z := 1]$
   But there is no z in the body of the function, so there is nowhere to put a 1. We eliminate the head, and the final result is:

9. a

It's more common in academic lambda calculus materials to refer to abstract variables rather than concrete values. The process of beta reduction is the same, regardless. The lambda calculus is a process or method, like a game with a few simple rules for transforming lambdas but no specific meaning. We've introduced concrete values to make the reduction somewhat easier to see.

The next example uses only abstract variables. Due to alpha equivalence, you sometimes see expressions in lambda calculus literature such as:

$$(\lambda xy.xxy)(\lambda x.xy)(\lambda x.xz)$$

The substitution process can become a tangle of x variables that are not the same x, because each is bound by a different head. To help make the reduction easier to read, we're going to use different variables in each abstraction, but it's worth emphasizing that the name of the variable (the letter) has no meaning or significance:

1. $(\lambda xyz.xz(yz))(\lambda mn.m)(\lambda p.p)$

2. $(\lambda x.\lambda y.\lambda z.xz(yz))(\lambda m.\lambda n.m)(\lambda p.p)$
   We haven't reduced or applied anything here, only made the currying explicit.

3. $(\lambda y.\lambda z.(\lambda m.\lambda n.m)z(yz))(\lambda p.p)$
   Our first reduction step was to apply the outermost lambda, which was binding the x, to the first argument, $(\lambda m.\lambda n.m)$.

4. $\lambda z.(\lambda m.\lambda n.m)(z)((\lambda p.p)z)$
   We applied the y and replaced the single occurrence of y with the next argument, the term $\lambda p.p$. The outermost lambda binding z is, at this point, irreducible, because it has no argument to apply to. What remains is to go inside the terms one layer at a time until we find something reducible.

5. $\lambda z.(\lambda n.z)((\lambda p.p)z)$
   We can apply the lambda binding m to the argument z. We keep searching for terms we can apply. The next thing we can apply is the lambda binding n to the lambda term $((\lambda p.p)z)$.

6. $\lambda z.z$
   In the final step, the reduction takes a turn that might look slightly odd. Here, the outermost, leftmost ***reducible*** term is $\lambda n.z$ applied to the entirety of $((\lambda p.p)z)$. As we saw in an example above, it doesn't matter what n got bound to—$\lambda n.z$ unconditionally tosses the argument and returns z. So, we are left with an irreducible lambda expression.

Intermission: Equivalence exercises

We'll give you a lambda expression. Keeping in mind both alpha equivalence and how multiple heads are nested, choose an answer that is equivalent to the listed lambda term:

1. $\lambda xy.xz$
   a) $\lambda xz.xz$
   b) $\lambda mn.mz$
   c) $\lambda z.(\lambda x.xz)$

2. $\lambda xy.xxy$
   a) $\lambda mn.mnp$
   b) $\lambda x.(\lambda y.xy)$
   c) $\lambda a.(\lambda b.aab)$

3. $\lambda xyz.zx$
   a) $\lambda x.(\lambda y.(\lambda z.z))$
   b) $\lambda tos.st$
   c) $\lambda mnp.mn$

1.7 Evaluation is simplification

There are multiple normal forms in lambda calculus, but when we refer to normal form here, we mean *beta normal form*. Beta normal form is when you cannot beta reduce (apply lambdas to arguments) the terms any further. This corresponds to a fully evaluated expression, or, in programming, a fully executed program. This is important, because it's how you know that you're done evaluating an expression. It's also valuable to have an appreciation for evaluation as a form of simplification when you get to Haskell code, as well.

Don't be intimidated by calling the reduced form of an expression its normal form. When you want to say 2, do you say 2000 / 1000 each time, or do you say 2? The expression 2000 / 1000 is not fully evaluated. The division function has been fully applied to two arguments, but it hasn't yet been reduced or evaluated. In other words, there's a simpler form it can be reduced to—the number two. The normal form, therefore, is 2.

The point is that if you have a function, such as division—or /—and that function is "saturated" (it's been applied to all of its arguments), but you haven't yet simplified it to the final result, then it is not fully evaluated, only applied. Application is what makes evaluation/simplification possible.

Similarly, the normal form of the following is 600:

$$(10 + 2) * 100/2$$

We cannot reduce the number 600 any further. There are no more functions that we can beta reduce. Normal form means there is nothing left that can be reduced.

The identity function, $\lambda x.x$, is fully reduced (that is, in normal form), because it hasn't yet been applied to anything. However, $(\lambda x.x)z$ is ***not*** in beta normal form, because the identity function has been applied to a free variable z and hasn't been reduced. If we did reduce it, the final result in beta normal form would be z.

1.8 Combinators

A combinator is a lambda term with no free variables. Combinators, as the name suggests, serve only to *combine* the arguments they are

given.

So, the following are combinators, because every term in the body occurs in the head:

1. $\lambda x.x$
   x is the only variable and is bound, because it is bound by the enclosing lambda.

2. $\lambda xy.x$

3. $\lambda xyz.xz(yz)$

And the following are not, because they contain free variables:

1. $\lambda y.x$
   Here, y is bound (it's in the head of the lambda), but x is free.

2. $\lambda x.xz$
   x is bound and is used in the body, but z is free.

We won't have a lot to say about combinators, per se. The point is to call out a special class of lambda expressions that can *only* combine the arguments they are given, without introducing any new values or random data.

1.9 Divergence

Not all reducible lambda terms reduce to a normal form. This isn't because they're already fully reduced, but because they *diverge*. Divergence here means that the reduction process never terminates or ends. Reducing terms should ordinarily *converge* to beta normal form, and divergence is the opposite of convergence, or normal form. Here's an example of a lambda term called *omega* that diverges:

1. $(\lambda x.xx)(\lambda x.xx)$
   x in the first lambda's head becomes the second lambda.

2. $([x := (\lambda x.xx)]xx)$
   Using $[var := expr]$ to denote what x has been bound to.

3. $(\lambda x.xx)(\lambda x.xx)$
   Substituting $(\lambda x.xx)$ for each occurence of x. We're back to where we started, and this reduction process never ends—we can say in this case that omega diverges.

This matters in programming, because terms that diverge are terms that don't produce an *answer* or meaningful result. Understanding what will terminate means understanding what programs will do useful work and return the answer we want. We'll cover this idea later.

1.10 Summary

The main points you should take away from this chapter are:

- Functional programming is based on expressions that include variables or constant values, expressions combined with other expressions, and functions.
- Functions have a head and a body and are those expressions that can be applied to arguments and reduced, or evaluated, to a final result.
- Variables may be bound in the function declaration, and every time a bound variable shows up in a function, it has the same value.
- All functions take one argument and return one result.
- Functions are a mapping of a set of inputs to a set of outputs. Given the same input, they always return the same result.

These things all apply to Haskell, as they do to any purely functional language, because, semantically, Haskell is a lambda calculus. Actually, Haskell is a *typed* lambda calculus—more on types later—with a lot of surface-level decoration sprinkled on top, to make it easier for humans to write, but the semantics of the core language are the same as the lambda calculus. That is, the meaning of Haskell programs is centered around evaluating expressions rather than executing instructions, although Haskell has a way to execute instructions, too. We will still be making reference to the lambda calculus

when we come later to the apparently very complex topics: function composition, monads, parser combinators, etc. Don't worry if you don't know those words yet. If you understood this chapter, you have the foundation you need to understand them all.

1.11 Chapter exercises

We're going to do the following exercises differently than what you'll see in the rest of the book. Unlike in subsequent chapters, here we provide some answers and explanations for the questions below.

Combinators Determine whether each of the following functions are combinators or not:

1. $\lambda x.xxx$
2. $\lambda xy.zx$
3. $\lambda xyz.xy(zx)$
4. $\lambda xyz.xy(zxy)$
5. $\lambda xy.xy(zxy)$

Normal form or diverge? Determine whether each of the following expressions can be reduced to a normal form or if they diverge:

1. $\lambda x.xxx$
2. $(\lambda z.zz)(\lambda y.yy)$
3. $(\lambda x.xxx)z$

Beta reduce Evaluate (that is, beta reduce) each of the following expressions to normal form. We *strongly* recommend writing out the steps on paper with a pen or pencil:

1. $(\lambda abc.cba)zz(\lambda wv.w)$
2. $(\lambda x.\lambda y.xyy)(\lambda a.a)b$
3. $(\lambda y.y)(\lambda x.xx)(\lambda z.zq)$

4. $(\lambda z.z)(\lambda z.zz)(\lambda z.zy)$
   Hint: alpha equivalence.

5. $(\lambda x.\lambda y.xyy)(\lambda y.y)y$

6. $(\lambda a.aa)(\lambda b.ba)c$

7. $(\lambda xyz.xz(yz))(\lambda x.z)(\lambda x.a)$

1.12 Answers

Please note: this is the only chapter in the book for which we have provided answers. We do so, because it is important that you are able to check your understanding of this material and also due to the relative difficulty of checking answers that you probably wrote by hand in a notebook.

Equivalence exercises

1. b
2. c
3. b

Combinators

1. $\lambda x.xxx$ is indeed a combinator. The function refers only to the variable x, which is introduced as an argument.
2. $\lambda xy.zx$ is not a combinator. The variable z was not introduced as an argument and is thus a free variable.
3. $\lambda xyz.xy(zx)$ is a combinator, as all terms are bound. The head is λxyz, and the body is $xy(zx)$. None of the arguments in the head have been applied, so it's irreducible. The variables x, y, and z are all bound in the head and are not free. This makes the lambda a combinator—there are no occurrences of free variables.
4. $\lambda xyz.xy(zxy)$ is a combinator. The lambda has as its head λxyz, and its body is $xy(zxy)$. Again, none of the arguments have been applied, so it's irreducible. All that is different is that the bound variable y is referenced twice rather than once. There are still no free variables, so this is also a combinator.
5. $\lambda xy.xy(zxy)$ is not a combinator, because z is free. Note that z isn't bound in the head.

Normal form or diverge?

1. $\lambda x.xxx$ doesn't diverge and has no further reduction steps. If it had been applied to itself, it ***would*** diverge, but by itself it does not, as it is already in normal form.

2. $(\lambda z.zz)(\lambda y.yy)$ diverges: it never reaches a point where the reduction is complete. This is the *omega* term we showed you earlier, with different names for the bindings. It's *alpha-equivalent* to $(\lambda x.xx)(\lambda x.xx)$.

3. $(\lambda x.xxx)z$ doesn't diverge, it reduces to zzz.

Beta reduce The following expressions are evaluated in *normal order*, whereby terms in the outer-most and left-most positions get evaluated (applied) first. This means that if all terms are in the outermost position (none are nested), then it's left-to-right application order:

1. $(\lambda abc.cba)zz(\lambda wv.w)$
   $(\lambda a.\lambda b.\lambda c.cba)(z)z(\lambda w.\lambda v.w)$
   $(\lambda b.\lambda c.cbz)(z)(\lambda w.\lambda v.w)$
   $(\lambda c.czz)(\lambda w.\lambda v.w)$
   $(\lambda w.\lambda v.w)(z)z$
   $(\lambda v.z)(z)$
   z

2. $(\lambda x.\lambda y.xyy)(\lambda a.a)b$
   $(\lambda y.(\lambda a.a)yy)(b)$
   $(\lambda a.a)(b)b$
   bb

3. $(\lambda y.y)(\lambda x.xx)(\lambda z.zq)$
   $(\lambda x.xx)(\lambda z.zq)$
   $(\lambda z.zq)(\lambda z.zq)$
   $(\lambda z.zq)(q)$
   qq

4. $(\lambda z.z)(\lambda z.zz)(\lambda z.zy)$
   $(\lambda z.zz)(\lambda z.zy)$
   $(\lambda z.zy)(\lambda z.zy)$

$(\lambda z.zy)(y)$
yy

5. $(\lambda x.\lambda y.xyy)(\lambda y.y)y$
   $(\lambda y.(\lambda y.y)yy)(y)$
   $(\lambda y.y)(y)y$
   yy

6. $(\lambda a.aa)(\lambda b.ba)c$
   $(\lambda b.ba)(\lambda b.ba)c$
   $(\lambda b.ba)(a)c$
   aac

7. Steps we took:

   a) $(\lambda xyz.xz(yz))(\lambda x.z)(\lambda x.a)$

   b) $(\lambda x.\lambda y.\lambda z.xz(yz))(\lambda x.z)(\lambda x.a)$

   c) $(\lambda y.\lambda z1.(\lambda x.z)z1(yz1))(\lambda x.a)$

   d) $(\lambda z1.(\lambda x.z)(z1)((\lambda x.a)z1))$

   e) $(\lambda z1.z((\lambda x.a)(z1)))$

   f) $(\lambda z1.za)$
   The $z1$ notation allows us to distinguish two variables named z that came from different places. One is bound by the first head; the second is a free variable in the second lambda expression.

   How we got there, step by step:

   a) The expression we'll reduce.

   b) Add the implied lambdas to introduce each argument.

   c) Apply the leftmost x, and bind it to $(\lambda x.z)$. Rename the leftmost z to $z1$ for clarity, to avoid confusion with the other z. Hereafter, "z" is exclusively the z in $(\lambda x.z)$.

   d) Apply y, it gets bound to $(\lambda x.a)$.

   e) Can't apply $z1$ to anything, and our evaluation strategy is normal order, so leftmost outermost is the order of the day. Our leftmost, outermost lambda has no remaining arguments to be applied, so we now examine the terms

nested within to see if they are in normal form. $(\lambda x.z)$ gets applied to $z1$, tosses the $z1$ away, and returns z. z is now being applied to $((\lambda x.a)(z1))$.

f) We cannot reduce z further—it's free, and we know nothing, so we go inside yet another nesting and reduce $((\lambda x.a)(z1))$. $\lambda x.a$ gets applied to $z1$ but tosses it away and returns the free variable a. The a is now part of the body of that expression. All of our terms are in normal order now.

1.13 Definitions

1. The *lambda* in lambda calculus is the greek letter λ used to introduce, or abstract, arguments for binding in an expression.

2. A lambda *abstraction* is an anonymous function or lambda term.

   $(\lambda x.x + 1)$

   The head of the expression, $\lambda x.$, abstracts out the term $x + 1$. We can apply it to any x and recompute different results for each x to which we apply the lambda.

3. *Application* is how one evaluates or reduces lambdas, binding the parameter to the concrete argument. The argument is what specific term the lambda is applied to. Computations are performed in lambda calculus by applying lambdas to arguments until you run out of applications to perform:

   $(\lambda x.x)1$

   This example reduces to 1, as the identity $\lambda x.x$ is applied to the value 1, x is bound to 1, and since the lambda's body is x, it just kicks the 1 out. In a sense, applying the $\lambda x.x$ ***consumes*** it. We *reduce* the amount of structure with which we start.

4. *Lambda calculus* is a formal system for expressing programs in terms of abstraction and application.

5. *Normal order* is a common evaluation strategy in lambda calculi. Normal order means evaluating the leftmost, outermost lambdas first, evaluating nested terms after you've run out of arguments to apply. Normal order isn't how Haskell code is evaluated. Haskell's evaluation strategy is *call-by-need*, instead.

We'll explain this later. Answers to the evaluation exercises are in normal order.

1.14 Follow-up resources

These references are *optional* and intended only to offer suggestions for how you might deepen your understanding of this topic. They are ordered, here and in subsequent chapters, approximately from most approachable to most thorough:

1. Raul Rojas. *A Tutorial Introduction to the Lambda Calculus*.
   `http://www.inf.fu-berlin.de/lehre/WS03/alpi/lambda.pdf`
2. Henk Barendregt and Erik Barendsen. *Introduction to Lambda Calculus*.
   `http://www.cse.chalmers.se/research/group/logic/TypesSS05/Extra/geuvers.pdf`
3. Jean-Yves Girard, P. Taylor, and Yves Lafon. *Proofs and Types*.
   `http://www.paultaylor.eu/stable/prot.pdf`

Chapter 2

Hello, Haskell!

> Functions are beacons of
> constancy in a sea of turmoil.
>
> Mike Hammond

2.1 Hello, Haskell

Welcome to your first step in learning Haskell. Before you begin with the main course, you will need to install the necessary tools in order to complete the exercises as you work through the book. At this time, we recommend that you install Stack, which will give you GHC Haskell, the interactive environment called GHCi, and a project build tool and dependency manager all at once.

You can find the installation instructions online at `http://docs.haskellstack.org/en/stable/README/`, and there is also great documentation that can help you get started using Stack. You can also find installation instructions at `https://github.com/bitemyapp/learnhaskell`; there you will find, in addition, advice on learning Haskell and links to more exercises that may supplement what you're doing with this book.

The rest of this chapter will assume that you have completed the installation and are ready to begin working. In this chapter, you will:

- Use Haskell code in the interactive environment and also from source files.
- Understand the building blocks of Haskell: expressions and functions.
- Learn some features of Haskell syntax and conventions of good Haskell style.
- Modify simple functions.

2.2 Interacting with Haskell code

Haskell offers two primary ways of working with code. The first is inputting it directly into the interactive environment known as GHCi, or the REPL. The second is typing it into a text editor, saving, and then loading that source file into GHCi. This section offers an introduction to each method.

Using the REPL

REPL is an acronym for *read-eval-print loop*. REPLs are interactive programming environments in which you can input code, have it

evaluated, and see the result. They originated with Lisp but are now common to many modern programming languages, including Haskell.

Assuming you've completed your installation, you should be able to open your terminal or command prompt, type `ghci` or `stack ghci`,[1] hit enter, and see something like the following:

```
GHCi, version 8.6.5:
  http://www.haskell.org/ghc/  :? for help
Prelude>
```

If you are a Mac user, the first time you try to run `ghci`, you may get an error that looks something like this:

```
C compiler cannot create executables
```

If this happens, you probably just need to accept XCode's EULA. You can enter this command in the terminal to do so:

```
$ sudo xcodebuild -license
```

If you're using `stack ghci`,[2] there is probably a lot more startup text, and the prompt might be something other than `Prelude`. That's all fine. To match what we use in the book, ensure your Stack snapshot is `lts-13.24` and that your GHC version is `8.6.5`. You can enforce those versions either by setting the `resolver` in your Stack global project[3] to `lts-13.24` or by running `stack ghci` in a directory containing a file named `stack.yaml` with the following contents:

```
# stack.yaml
resolver: lts-13.24
packages: []
```

Now, try entering some simple arithmetic at your prompt:

[1] If you have installed GHC outside of Stack, then you should be able to open it with just the `ghci` command, but if your only GHC installation is what Stack installed, then you will need to use `stack ghci`.

[2] At this point in the book, you don't need to use `stack ghci`, but in later chapters, when we're importing a lot of modules and building projects, it will be much more convenient.

[3] See `https://docs.haskellstack.org/en/stable/yaml_configuration/` for more about Stack configuration.

```
Prelude> 2 + 2
4
Prelude> 7 < 9
True
Prelude> 10 ^ 2
100
```

If you can enter simple equations at the prompt and get the expected results, congratulations—you are now a functional programmer! More to the point, your REPL is working well, and you are ready to proceed.

To exit GHCi, use the command `:quit` or `:q`.

What is Prelude? `Prelude` is a library of standard functions. Opening GHCi or Stack GHCi automatically loads those functions so they can be used without your needing to do anything special. You can turn `Prelude` off, as we will show you later, and there are alternative preludes, though we won't use them in this book. `Prelude` is contained in Haskell's `base` package, which can be found at `https://www.stackage.org/package/base`. You'll see us mention sometimes that something or other is "in `base`," which means that it's contained in that large standard package.

GHCi commands

Throughout the book, we'll be using GHCi commands, such as `:quit` and `:info` in the REPL. Special commands that only GHCi understands begin with the `:` character. `:quit` is *not* Haskell code; it's just a GHCi feature.

We will present them in the text spelled out, but they can generally be abbreviated to just the colon and the first letter. That is, `:quit` becomes `:q`, `:info` becomes `:i`, and so forth. It's good to type the word out the first few times you use it, to help you remember what the abbreviation stands for, but after a few mentions, we will only use the abbreviations.

Working from source files

As nice as REPLs are, usually you want to store code in a file, so you can build it incrementally. Almost all nontrivial programming

you do will involve editing libraries or applications made of nested directories containing files with Haskell code in them. The basic process is to have the code and imports (more on that later) in a file, load it into the REPL, and interact with it there as you're building, modifying, and testing it.

You'll need a file named `test.hs`. The `.hs` file extension denotes a Haskell source code file. Depending on your setup and the workflow you're comfortable with, you can make a file by that name and then open it in your text editor or you can open your text editor, open a new file, and then save the file with that file name.

Then, enter the following code into the file and save it:

```
sayHello :: String -> IO ()
sayHello x =
  putStrLn ("Hello, " ++ x ++ "!")
```

Here, `::` is a way to write down a type signature. You can think of it as saying *has the type*. So, `sayHello` has the type `String -> IO ()`. These first chapters are focused on syntax, and we'll talk about types in a later chapter.

Next, in the same directory where you've stored your `test.hs` file, open your `ghci` REPL and type the following:

```
Prelude> :load test.hs
Prelude> sayHello "Haskell"
Hello, Haskell!
Prelude>
```

After using `:load` to load your `test.hs` file, the `sayHello` function is visible in the REPL, and you can pass it a string argument, such as "Haskell" (note the quotation marks), and see the output.

You may notice that after loading code from a source file, the GHCi prompt is no longer `Prelude>`. We will not reflect those changes to the prompt in our REPL examples. There may be exceptions to this later when we demonstrate multi-line GHCi definitions. To return to the `Prelude>` prompt, use the command `:m`, which is short for `:module`. This will unload the file from GHCi, which means that the code in that file will no longer be available, or "in scope," in your REPL.

2.3 Understanding expressions

Everything in Haskell is an expression or declaration. *Expressions* may be values, combinations of values, and/or functions applied to values. Expressions evaluate to a result. In the case of a literal value, the evaluation is trivial, as it only evaluates to itself. In the case of an arithmetic equation, the evaluation process is the process of computing the operator and its arguments, as you might expect. But, even though not all of your programs will be about doing arithmetic, all of Haskell's expressions work in a similar way, evaluating to a result in a predictable, transparent manner. Expressions are the building blocks of our programs, and programs themselves are one big expression made up of smaller expressions.

Regarding *declarations*, it suffices to say for now that they are top-level bindings that allow us to name expressions. We can then use those names to refer to them multiple times without copying and pasting the expressions.

The following are all expressions:

```
1
1 + 1
"Icarus"
```

Each can be examined in the GHCi REPL by entering the code at the prompt, then hitting "enter" to see the result of evaluating the expression. The numeric value `1`, for example, has no further reduction step, so it stands for itself.

If you haven't already, open up your terminal and get your REPL going to start following along with the code examples.

When we enter this into GHCi:

```
Prelude> 1
1
```

We see 1 printed, because it cannot be reduced any further.

In the next example, GHCi reduces the expression `1 + 2` to 3, then prints the number 3. The reduction terminates with the value 3, because there are no more terms to evaluate:

```
Prelude> 1 + 2
3
```

Expressions can be nested in numbers limited only by our willingness to take the time to write them down, much like in arithmetic:

```
Prelude> (1 + 2) * 3
9
Prelude> ((1 + 2) * 3) + 100
109
```

You can keep expanding on this, nesting as many expressions as you'd like and evaluating them. But, we don't have to limit ourselves to expressions such as these.

Normal form We say that expressions are in *normal form* when there are no more evaluation steps that can be taken, or, put differently, when they've reached an irreducible form. The normal form of `1 + 1` is 2. Why? Because the expression `1 + 1` can be evaluated or reduced by applying the addition operator to the two arguments. In other words, `1 + 1` is a reducible expression, while 2 is an expression but is no longer reducible—it can't evaluate into anything other than itself. Reducible expressions are also called *redexes*. While we will generally refer to this process as evaluation or reduction, you may also hear it called "normalizing" or "executing" an expression, although these are somewhat imprecise.

2.4 Functions

Expressions are the most basic units of a Haskell program, and a *function* is a specific type of expression. Functions in Haskell are related to functions in mathematics, which is to say they map an input or set of inputs to an output. A function is an expression that is applied to an argument and always returns a result. Because they are built purely of expressions, they will always evaluate to the same result when given the same values.

As in the lambda calculus, all functions in Haskell take one argument and return one result. The way to think of this is that, in Haskell, when it seems like we are passing multiple arguments to a function, we are actually applying a series of nested functions, each to one argument. This is called *currying*.

You may have noticed that the expressions we've looked at so far use literal values with no variables or abstractions. Functions allow us to abstract the parts of code we want to reuse for different literal values. Instead of nesting addition expressions, for example, we could write a function that would add the value we want wherever we call that function.

For example, say you had a bunch of expressions you needed to multiply by 3. You could keep entering them as individual expressions like this:

```
Prelude> (1 + 2) * 3
9
Prelude> (4 + 5) * 3
27
Prelude> (10 + 5) * 3
45
```

But you don't want to do that. Functions are how we factor out a pattern into something we can reuse with different inputs. You do that by naming the function and introducing an independent variable as the argument to the function. Functions can also appear in the expressions that form the bodies of other functions or be used as arguments to functions, just as any other value can be.

In this case, we have a series of expressions that we want to multiply by 3. Let's think in terms of a function: which part is common to all of the expressions? Which part varies? We know we have to give functions a name and apply them to an argument, so what could we call this function and what sort of argument might we apply it to?

The common pattern is the `* 3` bit. The part that varies is the addition expression before it, so we will make that a variable. We will name our function and apply it to the variable. When we input a value for the variable, our function will evaluate that, multiply it by 3, and return a result. In the next section, we will formalize this into a Haskell function.

Defining functions

Function definitions all share a few things in common. First, they start with the name of the function. This is followed by the formal

parameters[4] of the function, separated only by white space. Next there is an equals sign, which expresses equality of the terms. Finally, there is an expression that is the body of the function and that can be evaluated to return a value.

Defining functions in a Haskell source code file and in GHCi used to be a little different. To introduce definitions of values or functions in GHCi, you previously had to prefix the declaration with `let`.[5] Now, you don't need `let`, so your declaration can look like this in the REPL:

```
Prelude> triple x = x * 3
```

In a source file, it should be identical save for the absence of a REPL prompt, looking like this:

```
triple x = x * 3
```

Let's examine each part of this code:

```
  triple x   =   x * 3
-- [1]  [2] [3]  [ 4 ]
```

1. This is the name of the function we are defining; it is a function *declaration*. Note that it begins with a lowercase letter.
2. This is the parameter of the function. The parameters of our function correspond to the head of a lambda and bind variables that appear in the body expression.
3. The `=` is used to define (or *declare*) values and functions. This is *not* how we test for equality between two values in Haskell.
4. This is the body of the function, an expression that could be evaluated if the function is applied to a value. If `triple` is applied, the argument it's applied to will be the value to which the `x` is

[4] In practice, the terms *argument* and *parameter* are often used interchangeably, but there is a difference. *Argument* properly refers to the value or values passed to the function's *parameters* when the function is applied, not to the variables that represent them in the function definition (or those in the type signature). See the definitions at the end of this chapter for more information.

[5] This changed as of the release of GHC 8.0.1—using `let` in declarations in GHCi is no longer necessary. As we assume that most readers of this edition will be using a newer version of GHC, we have not used `let` notation in this book. This shouldn't cause any errors or breakage. If you're using an older version of GHC, add a `let` before your declarations in GHCi.

bound. Here, the expression `x * 3` constitutes the body of the function. So, if you have an expression like `triple 6`, `x` is bound to `6`. Since you've applied the function, you can also replace the fully applied function with its body and bound arguments.

Capitalization matters! Function names start with lowercase letters. Sometimes, for the sake of clarity, you may prefer camelCase style, and that is good style provided the first letter remains lowercase.

Variables must also begin with lowercase letters. They need not be single letters.

Playing with the triple function First, try entering the `triple` function directly into the REPL. Now, call the function by name, and introduce a numeric value for the `x` argument:

```
Prelude> triple 2
6
```

Next, enter it into a source file, and save the file. Load it into GHCi, using the `:load` or `:l` command. Once it's loaded, you can call the function at the prompt using the function name, `triple`, followed by a numeric value, just as you did in the REPL example above. Try using different values for `x`—integer values or other arithmetic expressions. Then try changing the function itself in the source file and reloading it to see what changes. You can use `:reload`, or `:r`, to reload the same file.

2.5 Evaluation

When we talk about evaluating an expression, we're talking about reducing the terms until the expression reaches its simplest form. Once a term has reached its simplest form, we say that it is *irreducible* or finished evaluating. Usually, we call this a value. Haskell uses a non-strict evaluation (sometimes called "lazy evaluation") strategy that defers evaluation of terms until they're forced to evaluate by other terms that refer to them.

Values are irreducible, but applications of functions to arguments are reducible. Reducing an expression means evaluating the terms

until you're left with a value. As in the lambda calculus, application is evaluation: applying a function to an argument allows evaluation or reduction.

Values are expressions, but they cannot be reduced further. Values, in other words, are a terminal point of reduction:

```
1
"Icarus"
```

The following expressions can be reduced to a value:

```
1 + 1
2 * 3 + 1
```

Each can be evaluated in the REPL, which reduces the expressions and then prints what they reduce to.

Let's get back to our `triple` function. Calling the function by name and applying it to an argument makes it a reducible expression. In a pure functional language like Haskell, we can replace applications of functions with their definitions and get the same results, like in math. As a result, when we see:

```
triple 2
```

Since triple is defined as `x = x * 3`, we know the expression is equivalent to:

```
triple 2
-- [triple x = x * 3; x:= 2]
2 * 3
6
```

We've applied `triple` to the value 2 and then reduced the expression to the final result 6. Our expression `triple 2` is in canonical or *normal form* when it reaches the number 6, because the value 6 has no remaining reducible expressions.

Haskell doesn't evaluate everything to canonical or normal form by default. Instead, it only evaluates to weak head normal form (WHNF) by default. What this means is that not everything will get reduced to its irreducible form immediately, so this expression:

```
(\f -> (1, 2 + f)) 2
```

Reduces to the following in WHNF:

```
(1, 2 + 2)
```

This representation is an approximation, but the key point here is that `2 + 2` is not evaluated to 4 until the last possible moment.

Exercises: Comprehension check

1. Given the following lines of code as they might appear in a source file:

   ```
   half x = x / 2

   square x = x * x
   ```

   Write the same declarations in your REPL, and then use the functions `half` and `square` in some experimental expressions.

2. Write one function that has one parameter and works for all the following expressions. Be sure to name the function.

   ```
   3.14 * (5 * 5)
   3.14 * (10 * 10)
   3.14 * (2 * 2)
   3.14 * (4 * 4)
   ```

3. There is a value in `Prelude` called `pi`. Rewrite your function to use `pi` instead of `3.14`.

2.6 Infix operators

Functions in Haskell default to prefix syntax, meaning that the function being applied is at the beginning of the expression rather than the middle. We saw that with our `triple` function, and we see it with standard functions such as the identity, or `id`, function. This function returns whatever value it is given as an argument:

```
Prelude> id 1
1
```

While this is the default syntax for functions, not all functions are prefix. There are a group of operators, such as the arithmetic operators we've been using, that are indeed functions (they apply to arguments to produce an output) but appear by default in an infix position.

Operators are functions that can be used in infix style. All operators are functions; not all functions are operators. While `triple` and `id` are prefix functions (*not* operators), the `+` function is an infix operator:

```
Prelude> 1 + 1
2
```

Now, we'll try a few other arithmetic operators:

```
Prelude> 100 + 100
200
Prelude> 768395 * 21356345
16410108716275
Prelude> 123123 / 123
1001.0
Prelude> 476 - 36
440
Prelude> 10 / 4
2.5
```

You can sometimes use functions infix style, with a small change in syntax:

```
Prelude> 10 `div` 4
2
Prelude> div 10 4
2
```

And you can use infix operators in prefix fashion by wrapping them in parentheses:

```
Prelude> (+) 100 100
200
Prelude> (*) 768395 21356345
```

```
16410108716275
Prelude> (/) 123123 123
1001.0
```

If the function name is alphanumeric, it is a prefix function by default, and not all prefix functions can be made infix. If the name is a symbol, it is infix by default but can be made prefix by wrapping it in parentheses.[6]

Associativity and precedence

As you may remember from your math classes, there's a default associativity and precedence to the infix operators `*`, `+`, `-`, and `/`.

We can ask GHCi for information such as associativity and precedence of operators and functions by using the `:info` command. When you ask GHCi for the `:info` about an operator or function, it provides the type information. It also tells you whether it's an infix operator, and, if it is, its associativity and precedence. Let's talk about that associativity and precedence briefly. We will elide the type information and so forth for now.

Here's what the code in `Prelude` says for `*`, `+`, and `-` at time of writing:

```
:info (*)
infixl  7   *
-- [1] [2] [3]

:info (+) (-)
infixl 6 +

infixl 6 -
```

1. `infixl` means it's an infix operator; the `l` means it's *left* associative.
2. 7 is the precedence: higher is applied first, on a scale of 0–9.
3. Infix function name: in this case, multiplication.

[6] For people who like nit-picky details: you cannot make a prefix function into an infix function using backticks, then wrap that in parentheses and make it into a prefix function. We're not clear why you'd want to do that anyway. Cut it out.

The information about addition and subtraction tells us they are both left-associative, infix operators with the same precedence (6).

Let's play with parentheses and see what it means that these associate to the left. Continue to follow along with the code by typing it into your REPL.

This:

```
2 * 3 * 4
```

Is evaluated as if it were:

```
(2 * 3) * 4
```

Because of left associativity.

Here's an example of a right-associative infix operator:

```
Prelude> :info (^)
infixr  8   ^
-- [1] [2] [3]
```

1. `infixr` means infix operator; the `r` means it's *right* associative.
2. 8 is the precedence. Higher precedence, indicated by higher numbers, is applied first, so this is higher precedence than multiplication (7), addition, or subtraction (both 6).
3. Infix function name: in this case, exponentiation.

Multiplication being associative means shifting the parentheses around in an expression doesn't change the result. Exponentiation is not associative. Thus, exponentiation is a prime candidate for demonstrating left vs. right associativity:

```
Prelude> 2 ^ 3 ^ 4
241785163922925834941235 2
Prelude> 2 ^ (3 ^ 4)
241785163922925834941235 2
Prelude> (2 ^ 3) ^ 4
4096
```

As you can see, adding parentheses starting from the right-hand side of the expression when the operator is right-associative doesn't change anything. However, if we parenthesize from the *left*, we get a different result when the expression is evaluated.

Your intuitions about precedence, associativity, and parenthesization from math classes will generally hold in Haskell:

```
2 + 3 * 4
```

```
(2 + 3) * 4
```

What's the difference between these two? Why are they different?

Exercises: Parentheses and association

Below are some pairs of functions that are alike except for parenthesization. Read them carefully and decide whether the parentheses change the results of the functions. Check your work in GHCi:

1. a) `8 + 7 * 9`
   b) `(8 + 7) * 9`
2. a) `perimeter x y = (x * 2) + (y * 2)`
   b) `perimeter x y = x * 2 + y * 2`
3. a) `f x = x / 2 + 9`
   b) `f x = x / (2 + 9)`

2.7 Declaring values

The order of declarations in a source code file doesn't matter, because GHCi loads the entire file at once, so it knows all the values that have been defined. On the other hand, when you enter them one by one into the REPL, the order does matter.

For example, we can declare a series of expressions in the REPL like this:

```
Prelude> y = 10
Prelude> x = 10 * 5 + y
Prelude> myResult = x * 5
```

We can now type the names of the values and hit enter to see their values:

```
Prelude> x
60
Prelude> y
10
Prelude> myResult
300
```

Let's see how to declare those values in a file called `learn.hs`. First, we declare the name of our module so it can be imported by name in a project (we won't be doing a project of this size for a while yet, but it's good to get in the habit of having module names):

```
-- learn.hs
module Learn where

x = 10 * 5 + y

myResult = x * 5

y = 10
```

Module names are capitalized. Also, in the variable name, we've used camelCase: the first letter is still lowercase, but we use an uppercase letter to delineate a word boundary for the sake of readability.

Troubleshooting

It is easy to make mistakes in the process of typing `learn.hs` into your editor. We'll look at a few common mistakes in this section. One thing to keep in mind is that the indentation of Haskell code is significant and varying it can potentially change its meaning. Incorrect indentation can also break your code. Use spaces, *not* tabs, to indent your source code.

In general, whitespace is significant in Haskell. Efficient use of whitespace makes the syntax more concise. This can take some getting used to if you've been working in another programming language. Whitespace is often the only mark of a function call, unless

parentheses are necessary due to conflicting precedence. Trailing whitespace, that is, extraneous whitespace at the end of a line of code, is considered bad style.

In source code files, indentation often replaces syntactic markers like curly braces, semicolons, and parentheses. The basic rule is that code that is part of an expression should be indented under the beginning of that expression, even when the beginning of the expression is not at the leftmost margin. Furthermore, parts of the expression that are grouped should be indented to the same level. For example, in a block of code introduced by `let` or `do`, you might see something like this:

```
let
  x = 3
  y = 4

-- or

let x = 3
    y = 4
```

This wouldn't work in a source file unless they were embedded in a top-level declaration.

Notice that the two definitions that are part of the expression line up in either case. It is incorrect to write:

```
let x = 3
 y = 4

-- or

let
 x = 3
  y = 4
```

If you have an expression that has multiple parts, your indentation will follow a pattern like this:

```
foo x =
    let y = x * 2
        z = x ^ 2
    in 2 * y * z
```

Notice that the definitions of `y` and `z` line up, and the definitions of `let` and `in` are also aligned. As you work through the book, pay attention to the indentation patterns we use. There are many cases where improper indentation will cause code not to work. Indentation can easily go wrong in a copy-and-paste job, as well.

If you make a mistake, like breaking up the declaration of `x`, such that the rest of the expression begins at the beginning of the next line:

```
-- Learn.hs

module Learn where
-- module declaration at the top

x = 10
* 5 + y

myResult = x * 5

y = 10
```

You might see an error like this:

```
Prelude> :l Learn.hs
[1 of 1] Compiling Learn

Learn.hs:7:1: error:
  parse error on input ‘*’
  |
4 | * 5 + y
  | ^
Failed, no modules loaded.
```

Note that the first line of the error message tells you where the error occurred: `:7:1` indicates that the mistake is in line 7, column 1, of the named file. That can make it easier to find the problem that needs to be fixed. Please note that the exact line and column numbers in your own error messages might be different from ours, depending on how you've entered the code into the file.

The way to fix this is to either put it all on one line, like this:

```
x = 10 * 5 + y
```

Or to make certain when you break up lines of code that the second line begins at least one space from the beginning of that line—either of the following should work:

```
x = 10
 * 5 + y

-- or

x = 10
    * 5 + y
```

The second one looks a little better. Generally, you should reserve the breaking up of lines for when you have code exceeding 100 columns in width.

Another possible error is not starting a declaration at the beginning (left) column of the line:

```
-- Learn.hs

module Learn where

 x = 10 * 5 + y

myResult = x * 5

y = 10
```

See that space before `x`? That will cause an error like:

```
Prelude> :l Learn.hs
[1 of 1] Compiling Learn

Learn.hs:7:1: error:
  parse error on input ‘myResult’
  |
5 | myResult = x * 5
  | ^^^^^^^^
Failed, no modules loaded.
```

This may confuse you, as `myResult` is not where you need to modify your code. The error is only an extraneous space, but all declarations in the module must start at the same column. The column that all declarations within a module must start in is determined by the first declaration in the module. In this case, the error message gives a location that is different from where you should fix the problem, because all the compiler knows is that the declaration of `x` made a single space the appropriate indentation for all declarations within that module, and the declaration of `myResult` began a column too early.

It is possible to fix this error by indenting the `myResult` and `y` declarations to the same level as the indented `x` declaration:

```
-- Learn.hs

module Learn where

 x = 10 * 5 + y

 myResult = x * 5

 y = 10
```

However, this is considered bad style and is not standard Haskell practice. There is seldom a good reason to indent all your declarations in this way, but noting this gives us some idea of how the compiler is reading the code. It is better, when confronted with an error message like this, to make sure that your first declaration is at the leftmost margin and proceed from there.

Another possible mistake is that you might've missed the second - in the -- used to comment out source lines of code.

So this code:

```
- Learn.hs

module Learn where

x = 10 * 5 + y

myResult = x * 5

y = 10
```

Will cause this error:

```
Learn.hs:3:1: error:
  parse error on input ‘module’
  |
3 | module Learn where
  | ^^^^^^
Failed, no modules loaded.
```

Note again that it says the parse error occurs at the beginning of the module declaration, but the issue is that the comment line, `- Learn.hs`, has only one dash, whereas it requires two to form a syntactically correct Haskell comment.

Now, we can see how to work with code that is saved in a source file from GHCi without manually copying and pasting the definitions into our REPL. Assuming we open our REPL in the same directory as we have `Learn.hs` saved, we can do the following:

```
Prelude> :load learn.hs
[1 of 1] Compiling Learn
Ok, one module loaded.
Prelude> x
60
Prelude> y
10
```

```
Prelude> myResult
300
```

Exercises: Heal the sick

The following code samples are broken and won't compile. The first two are as you might enter them into the REPL. The third is from a source file. Find the mistakes, and fix them so that the code will compile:

1. `area x = 3. 14 * (x * x)`
2. `double x = b * 2`
3. ```
   x = 7
    y = 10
   f = x + y
   ```

2.8 Arithmetic functions in Haskell

This section will explore basic arithmetic using some common operators and functions. We'll focus on the following subset of them:

Operator	Name	Purpose/application
`+`	plus	addition
`-`	minus	subtraction
`*`	asterisk	multiplication
`/`	slash	fractional division
`div`	divide	integral division, round down
`mod`	modulo	like `rem`, but after modular division
`quot`	quotient	integral division, round toward zero
`rem`	remainder	remainder after division

At the risk of stating the obvious, "integral" division refers to division of integers. Because it's integral and not fractional, it takes integers as arguments and returns integers as results. That's why the results are rounded.

Here's an example of each of the above operators in the REPL:
   ```

```
Prelude> 1 + 1
2
Prelude> 1 - 1
0
Prelude> 1 * 1
1
Prelude> 1 / 1
1.0
Prelude> div 1 1
1
Prelude> mod 1 1
0
Prelude> quot 1 1
1
Prelude> rem 1 1
0
```

You will usually want `div` for integral division, due to the way `div` and `quot` round:

```
-- rounds down
Prelude> div 20 (-6)
-4

-- rounds toward zero
Prelude> quot 20 (-6)
-3
```

Also, `rem` and `mod` have slightly different use cases; we'll look at `mod` in a little more detail later in this chapter. We will cover the `/` function in more detail in a later chapter, as that will require some explanation of types and type classes.

Laws for quotients and remainders

Programming often makes use of more division and remainder functions than standard arithmetic does, and it's helpful to be familiar with the laws governing the use of `quot` and `rem`, on the one hand, and `div` and `mod`, on the other.[7] We'll take a look at those here:

```
(quot x y)*y + (rem x y) == x

(div x y)*y + (mod x y) == x
```

We won't walk through a proof exercise, but we can demonstrate these laws a little bit:

```
(quot x y)*y + (rem x y)

Given x is 10 and y is (-4)

(quot 10 (-4))*(-4) + (rem 10 (-4))

quot 10 (-4) == (-2)  and  rem 10 (-4) == 2

(-2)*(-4) + (2) == 10

10 == x
```

Yes, we got to the result we wanted.

Now for `div` and `mod`:

```
(div x y)*y + (mod x y)

Given x is 10 and y is (-4)

(div 10 (-4))*(-4) + (mod 10 (-4))

div 10 (-4) == (-3)  and  mod 10 (-4) == -2

(-3)*(-4) + (-2) == 10

10 == x
```

Our result indicates that all is well in the world of integral division.

[7] From Lennart Augustsson's blog, `http://augustss.blogspot.com`, or the Stack Overflow answer at `http://stackoverflow.com/a/8111203`.

Using mod

This section is not a full discussion of modular arithmetic, but we want to give more direction on how to use `mod`, in general, for those who may be unfamiliar with it and how it works in Haskell, specifically.

We've already mentioned that `mod` gives the remainder of a modular division operation. If you're not already familiar with modular division, you may not understand the useful difference between `mod` and `rem`.

Modular arithmetic is a system of arithmetic for integers where numbers "wrap around" upon reaching a certain value, called the *modulus*. It is often explained in terms of a clock.

When we count time by a 12-hour clock, we have to wrap the counting around the 12. For example, if the time is now 8:00 and you want to know what time it will be 8 hours from now, you don't simply add 8 + 8 and get a result of 16 o'clock.[8]

Instead, you wrap the count around every 12 hours. So, adding 8 hours to 8:00 means that we add 4 hours to get to the 12, and at the 12 we start over again as if it's 0 and add the remaining 4 hours of our 8, for an answer of 4:00. That is, 8 hours after 8:00 is 4:00.

This is arithmetic *modulo* 12. In our 12-hour clock, 12 is equivalent to both itself and to 0, so the time at 12:00 is also, in some sense 0:00. Arithmetic modulo 12 means that 12 is both 12 and 0.

Often, this will give you the same answer that `rem` does:

```
Prelude> mod 15 12
3
Prelude> rem 15 12
3

Prelude> mod 21 12
9
Prelude> rem 21 12
9

Prelude> mod 3 12
```

[8] Obviously, with a 24-hour clock, such a time is possible. However, if we were starting from 8:00 p.m. and trying to find the time 8 hours later, the answer would not be 16:00 a.m. A 24-hour clock has a different modulus than a 12-hour clock.

```
3
Prelude> rem 3 12
3
```

If you're wondering what the deal is with the last two examples, it's because `mod` and `rem` can only represent integral division. If all you have to work with is integers, then dividing a smaller number by a larger number results in an answer of 0 with a remainder of whatever the smaller number (the dividend) is. If you want to divide a smaller number by a larger number and return a fractional answer, then you need to use `/`, and you won't have a remainder.

Let's say we need to write a function that will determine what day of the week it was or will be a certain number of days before or after this one. For our purposes here, we will assign a number to each day of the week, using 0 to represent Sunday.[9] Then, if today is Monday, and we want to know what day of the week it will be 23 days from now, we could do this:

```
Prelude> mod (1 + 23) 7
3
```

The 1 represents Monday, the current day, while 23 is the number of days we're trying to add. Using `mod` to wrap it around the 7 means it will return a number that corresponds to a day of the week in our numbering.

Likewise, if we add 5 days to Saturday, we get Thursday:

```
Prelude> mod (6 + 5) 7
4
```

We can use `rem` to do the same thing with apparently equivalent accuracy:

```
Prelude> rem (1 + 23) 7
3
```

However, if we want to *subtract* and find out what day of the week it was some number of days in the past, then we'll see a difference.

[9] Sure, you may naturally think of the days of the week as being numbered 1–7. But programmers like to index things from zero.

Let's try asking, if today is Wednesday (3), what day it was 12 days ago:

```
Prelude> mod (3 - 12) 7
5
Prelude> rem (3 - 12) 7
-2
```

The version with `mod` gives us the correct answer, while the `rem` version does not.

One key difference here is that, in Haskell (not in all languages), if one or both arguments are negative, the results of `mod` will have the same sign as the divisor, while the result of `rem` will have the same sign as the dividend:

```
Prelude> (-5) `mod` 2
1
Prelude> 5 `mod` (-2)
-1
Prelude> (-5) `mod` (-2)
-1
```

But:

```
Prelude> (-5) `rem` 2
-1
Prelude> 5 `rem` (-2)
1
Prelude> (-5) `rem` (-2)
-1
```

Figuring out when you need `mod` takes some experience.

Negative numbers

Due to the interaction of parentheses, currying, and infix syntax, negative numbers get special treatment in Haskell.

If you want a value that is a negative number by itself, this will work fine:

```
Prelude> -1000
-1000
```

However, this will not work in some cases:

```
Prelude> 1000 + -9

Precedence parsing error
    cannot mix ‘+’ [infixl 6] and
      prefix `-' [infixl 6] in the
        same infix expression
```

Fortunately, we are told about our mistake before any of our code is executed. Note how the error message tells you that the problem has to do with precedence. Addition and subtraction have the same precedence (6), and GHCi thinks we are trying to add and subtract, not add a negative number, so it doesn't know how to resolve the precedence and evaluate the expression. We need to make a small change before we can add a positive and a negative number together:

```
Prelude> 1000 + (-9)
991
```

The negation of numbers in Haskell by the use of a unary - sign is a form of *syntactic sugar*. Syntax is the grammar and structure of the text we use to express programs, and syntactic sugar is a means for us to make that text easier to read and write. Syntactic sugar can make the typing or reading of code nicer but changes nothing about the semantics, or meaning, of programs and doesn't change how we solve problems in code. Typically, when code with syntactic sugar is processed by our REPL or compiler, a simple transformation from the shorter ("sweeter") form to a more verbose, truer representation is performed after the code has been parsed.

In the specific case of -, the syntactic sugar means the operator now has two possible interpretations. The two possible interpretations of the syntactic - are either that - is being used as an alias for `negate` or that it is the subtraction function. The following are semantically identical (that is, they have the same meaning, despite different syntax) because the - is translated into `negate`:

```
Prelude> 2000 + (-1234)
766

Prelude> 2000 + (negate 1234)
766
```

Whereas this is - being used for subtraction:

```
Prelude> 2000 - 1234
766
```

Fortunately, syntactic overloading like this isn't common in Haskell.

2.9 Parenthesization

Here, we've listed the information that GHCi gives us for various infix operators. We have left the type signatures in, although it is not directly relevant at this time. This will give you a chance to look at the types if you're curious and also provide a more accurate picture of the `:info` command:

```
Prelude> :info (^)
(^) :: (Num a, Integral b) => a -> b -> a
infixr 8 ^

Prelude> :info (*)
class Num a where
  (*) :: a -> a -> a
infixl 7 *

Prelude> :info (+)
class Num a where
  (+) :: a -> a -> a
infixl 6 +

Prelude> :info (-)
class Num a where
  (-) :: a -> a -> a
infixl 6 -
```

```
Prelude> :info ($)
($) :: (a -> b) -> a -> b
infixr 0 $
```

In the case of `$`, you might see some cruft:

```
forall (r :: GHC.Types.RuntimeRep)
  a (b :: TYPE r).
```

Disregard this bit, and pretend it's solely the type described in the book.

We should take a moment to explain and demonstrate the `$` operator, as you will run into it fairly frequently in Haskell code. The good news is that it does almost nothing. The bad news is this fact sometimes trips people up.

First, here's the definition of `$`:

```
f $ a = f a
```

Immediately, this seems a bit pointless until we remember that it's defined as an infix operator with the lowest possible precedence. The `$` operator is a convenience for when you want to express something with fewer pairs of parentheses:

```
Prelude> (2^) (2 + 2)
16
-- can replace those parentheses
Prelude> (2^) $ 2 + 2
16
-- without either parentheses or $
Prelude> (2^) 2 + 2
6
```

The `$` function evaluates everything to its right first and can thereby be used to delay function application. You'll see what we mean about delaying function application, in particular, when we get to Chapter 7 and use it with function composition.

Also, note that you can stack up multiple uses of `$` in the same expression. For example, this works:

```
Prelude> (2^) $ (+2) $ 3*2
256
```

But this does not:

```
Prelude> (2^) $ 2 + 2 $ (*30)
```

A rather long and ugly type error about trying to use numbers as if they were functions follows. We can see why this code doesn't make sense if we examine the reduction steps:

```
-- Remember the definition of $
f $ a = f a

(2^) $ 2 + 2 $ (*30)
```

Given the right-associativity (`infixr`) of `$`, we must begin at the right-most position:

```
2 + 2 $ (*30)
-- reduce ($)
(2 + 2) (*30)
```

Then, we must evaluate `(2 + 2)` before we can apply it:

```
4 (*30)
```

You might think that this could evaluate as (4 * 30), but it's trying to apply 4 as if it were a function to the argument `(*30)`! Writing expressions like `(*30)` is called *sectioning*.

Now, let's flip that expression around a bit so it works and then walk through a reduction:

```
(2^) $ (*30) $ 2 + 2
-- must evaluate right-side first

(2^) $ (*30) $ 2 + 2
-- application of the function (*30) to the
-- expression (2 + 2) forces evaluation
```

```
(2^) $ (*30) 4
-- then we reduce (*30) 4
(2^) $ 120

-- reduce ($) again.
(2^) 120
-- reduce (2^)
1329227995784915872903807060280344576
```

Some Haskellers find parentheses more readable than the dollar sign, but it's too common in idiomatic Haskell code for you to not at least be familiar with it.

Parenthesizing infix operators

There are times when you want to refer to an infix function without applying any arguments, and there are also times when you want to use them as prefix operators instead of infix. In both cases, you must wrap the operator in parentheses. Let's look at how we use infix operators as prefixes.

If your infix function is `>>`, then you must write `(>>)` to refer to it as a value. `(+)` is the addition infix function without any arguments applied yet, and `(+1)` is the same addition function but with one argument applied, making it return the next argument it's applied to plus one:

```
Prelude> 1 + 2
3
Prelude> (+) 1 2
3
Prelude> (+1) 2
3
```

The last case is known as *sectioning* and allows you to pass around partially applied functions. With commutative functions, such as addition, it makes no difference if you use `(+1)` or `(1+)`, because the order of the arguments won't change the result.

If you use sectioning with a function that is not commutative, the order matters:

```
Prelude> (1/) 2
0.5
Prelude> (/1) 2
2.0
```

Subtraction, -, is a special case. These will work:

```
Prelude> 2 - 1
1
Prelude> (-) 2 1
1
```

The following, however, won't work:

```
Prelude> (-2) 1
```

Enclosing a value inside the parentheses with the - operator indicates to GHCi that it's the argument of a function. Because the - function represents negation, not subtraction, when it's applied to a single argument, GHCi does not know what to do with it, and so it returns an error message. Here, - is a case of syntactic overloading disambiguated by how it is used.

You can use sectioning for subtraction, but it must be the first argument:

```
Prelude> x = 5
Prelude> y = (1 -)
Prelude> y x
-4
```

Or, instead of `(- x)`, you can write `(subtract x)`:

```
Prelude> (subtract 2) 3
1
```

It may not be immediately obvious why you would ever want to do this, but you will see this syntax used throughout this book, for example, once we start wanting to apply functions to each value inside a list or other data structure.

2.10 let and where

You will often see `let` and `where` used to introduce components of expressions, and they seem similar. It takes some practice to get used to the appropriate times to use each.

The contrast here is that `let` introduces an *expression*, so it can be used wherever you can have an expression, but `where` is a *declaration* and is bound to a surrounding syntactic construct.

We'll start with an example of `where`:

```
-- FunctionWithWhere.hs
module FunctionWithWhere where

printInc n = print plusTwo
  where plusTwo = n + 2
```

And, if we use this in the REPL:

```
Prelude> :l FunctionWithWhere.hs
[1 of 1] Compiling FunctionWithWhere
Ok, one module loaded.
Prelude> printInc 1
3
```

Now, we have the same function but using `let` in the place of `where`:

```
-- FunctionWithLet.hs
module FunctionWithLet where

printInc2 n = let plusTwo = n + 2
              in print plusTwo
```

When you see `let` followed by `in`, you're looking at a *let expression*. Here's that function in the REPL:

```
Prelude> :load FunctionWithLet.hs
[1 of 1] Compiling FunctionWithLet
Ok, one module loaded.
Prelude> printInc2 3
5
```

If you loaded `FunctionWithLet` in the same REPL session as `FunctionWithWhere`, then it will have unloaded the first one before loading the new one:

```
Prelude> :load FunctionWithWhere.hs
[1 of 1] Compiling FunctionWithWhere
Ok, one module loaded.
Prelude> printInc 1
3

Prelude> :load FunctionWithLet.hs
[1 of 1] Compiling FunctionWithLet
Ok, one module loaded.
Prelude> printInc2 10
12
Prelude> printInc 10

• Variable not in scope:
    printInc :: Integer -> t
• Perhaps you meant ‘printInc2’ (line 5)
```

`printInc` isn't in scope anymore, because GHCi unloaded everything you'd defined or loaded after you used `:load` to load the `FunctionWithLet.hs` source file. *Scope* is the area of source code where a binding of a variable applies.

That is one limitation of the `:load` command in GHCi. As we build larger projects that require having multiple modules in scope, we will use a project manager called Stack rather than GHCi itself.

Exercises: A head code

Now for some exercises. First, determine in your head what the following expressions will return, then validate your assumptions in the REPL. These examples are prefixed with `let`, because they are not declarations, they are expressions:

1. `let x = 5 in x`
2. `let x = 5 in x * x`
3. `let x = 5; y = 6 in x * y`

4. `let x = 3; y = 1000 in x + 3`

Above, you entered some `let` expressions into your REPL to evaluate them. Now, we're going to open a file and rewrite some `let` expressions using `where` declarations. You will have to give the value you're binding a name, although the name can be a single letter if you like. For example, this:

```
-- this should work in GHCi
let x = 5; y = 6 in x * y
```

Could be rewritten as:

```
-- practice.hs
module Mult1 where


-- put this in a file
mult1      = x * y
   where x = 5
         y = 6
```

Making the equals signs line up is a stylistic choice. As long as things are nested in that way, the equals signs do not have to line up. But notice that we use a name to refer to this value in the REPL:

```
Prelude> :l practice.hs
[1 of 1] Compiling Main
Ok, one module loaded.
Prelude> mult1
30
```

The prompt changes to `*Main` instead of `Prelude` to indicate that you have a module called `Main` loaded.

Rewrite with `where` clauses:

1. `let x = 3; y = 1000 in x * 3 + y`

2. `let y = 10; x = 10 * 5 + y in x * 5`

3.
```
let x = 7
    y = negate x
    z = y * 10
in z / x + y
```

Note: the filename you choose is unimportant except for the `.hs` extension.

2.11 Chapter exercises

The goal for all of the following exercises is to get you playing with code and forming hypotheses about what it should do. Read the code carefully, using what we've learned so far. Generate a hypothesis about what you think the code will do. Play with it in the REPL, and find out where you were right or wrong.

Parenthesization

Given what we know about the precedence of `*`, `+`, and `^`, how can we parenthesize the following expressions more explicitly without changing their results? Put together answers you think are correct, then test them in the GHCi REPL.

For example, we want to make this more explicit:

```
2 + 2 * 3 - 3
```

This will produce the same result:

```
2 + (2 * 3) - 3
```

Attempt the above on the following expressions:

1. `2 + 2 * 3 - 1`
2. `(^) 10 $ 1 + 1`
3. `2 ^ 2 * 4 ^ 5 + 1`

Equivalent expressions

Which of the following pairs of expressions will return the same result when evaluated? Try to reason them out by reading the code, and then enter them into the REPL to check your work:

1. ```
   1 + 1
   ```

   ```
   2
   ```

2. ```
   10 ^ 2
   ```

   ```
   10 + 9 * 10
   ```

3. ```
   400 - 37
   ```

   ```
   (-) 37 400
   ```

4. ```
   100 `div` 3
   ```

   ```
   100 / 3
   ```

5. ```
   2 * 5 + 18
   ```

   ```
   2 * (5 + 18)
   ```

More fun with functions

Here is a bit of code as it might be entered into a source file. Remember that when you write code in a source file, the order is unimportant, but when writing code directly into the REPL, the order does matter. Given that, look at this code, and rewrite it such that it could be evaluated in the REPL. Be sure to enter your code into the REPL to make sure it evaluates correctly:
   ```

```
z = 7

x = y ^ 2

waxOn = x * 5

y = z + 8
```

1. Now, you have a value called `waxOn` in your REPL. What do you think will happen if you enter:

   ```
   10 + waxOn
   -- or
   (+10) waxOn
   -- or
   (-) 15 waxOn
   -- or
   (-) waxOn 15
   ```

2. Earlier, we looked at a function called `triple`. While your REPL has `waxOn` in session, re-enter the `triple` function at the prompt:

   ```
   triple x = x * 3
   ```

3. Now, what will happen if we enter this at our GHCi prompt? What do you think will happen first, considering what role `waxOn` is playing in this function call? Then enter it, see what does happen, and check your understanding:

   ```
   triple waxOn
   ```

4. Rewrite `waxOn` as an expression with a `where` clause in your source file. Load it into your REPL, and make sure it still works as expected.

5. To the same source file where you have `waxOn`, add the `triple` function. Remember: the function name should be at the left margin (that is, not nested as one of the `waxOn` expressions). Make sure it works by loading it into your REPL and then entering `triple waxOn` again at the REPL prompt. You should have the same answer as you did above.

6. Now, without changing what you've done so far in that file, add a new function called `waxOff` that looks like this:

   ```
   waxOff x = triple x
   ```

7. Load the source file into your REPL, and enter `waxOff waxOn` at the prompt.

   You now have a function, `waxOff`, that can be applied to a variety of arguments—not just `waxOn` but any (numeric) value you want to substitute for `x`. Play with that a bit. What is the result of `waxOff 10` or `waxOff (-50)`? Try modifying your `waxOff` function to do something new—perhaps you want to first triple the `x` value and then square it or divide it by 10. Spend some time getting comfortable with modifying the source file code, reloading it, and checking your modifications in the REPL.

2.12 Definitions

1. The terms *argument* and *parameter* are often used interchangeably. However, it is worthwhile to understand the distinction. A *parameter*, or formal parameter, *represents* a value that will be passed to a function when that function is called. Thus, parameters are usually variables. An *argument* is an input value the function is applied to. A function's parameter is bound to an argument when the function is applied to that argument. For example, in `f x = x + 2`, which takes an argument and returns that value added to 2, `x` is the one parameter of our function. We run the code by applying `f` to some argument. If the argument we pass to the parameter `x` is 2, our result would be 4. However, arguments can themselves be variables or be expressions that include variables, thus the distinction is not always clear. When we use "parameter" in this book, it will always refer to formal parameters, usually in a type signature, but we've taken the liberty of using "argument" somewhat more loosely.

2. An *expression* is a combination of symbols that conforms to syntactic rules and can be evaluated to some result. In Haskell, an expression is a well-structured combination of constants, variables, and functions. While irreducible constants are technically

expressions, we usually refer to those as "values," so we usually mean "reducible expression" when we use the term *expression*.

3. A *value* is an expression that cannot be reduced or evaluated any further. `2 * 2` is an expression, but not a value, whereas what it evaluates to, 4, is a value.

4. A *function* is a mathematical object that can be applied to an argument in order to return a result—and that's it. We can also describe a function as a list of ordered pairs of its inputs and the resulting outputs, like a mapping. Given the function `f x = x + 2` applied to the argument 2, we would have the ordered pair `(2, 4)` of its input and output.

5. *Infix notation* is the style used in arithmetic and logic. Infix means that the operator is placed between the operands or *arguments*. An example would be the plus sign in an expression like `2 + 2`.

6. *Operators* are functions that are infix by default. In Haskell, operators must use symbols and not alphanumeric characters.

7. *Syntactic sugar* is syntax within a programming language designed to make expressions easier to read and write.

2.13 Follow-up resources

1. Haskell Wiki. *Let vs. Where*.
   `https://wiki.haskell.org/Let_vs._Where`

2. Gabriel Gonzalez. *How to desugar Haskell code*.

Chapter 3

Strings

> Like punning, programming is a play on words.
>
> Alan Perlis

3.1 Printing strings

So far, we've been doing arithmetic using simple expressions. In this chapter, we will turn our attention to a different type of data called `String`.

Most programming languages refer to the data structures used to contain text as "strings," usually represented as sequences, or lists, of characters. In this section, we will:

- Take an introductory look at types to understand the data structure called `String`.
- Talk about the special syntax, or syntactic sugar, used for strings.
- Print strings in the REPL environment.
- Work with some standard functions that operate on this datatype.

3.2 A first look at types

First, since we will be working with strings, we want to start by understanding what these data structures are in Haskell as well as a bit of the special syntax we use for them. We haven't talked much about types yet, although you saw some examples of them in the last chapter. Types are important in Haskell, and the next two chapters are entirely devoted to them.

Types are a way of categorizing values. There are several types for numbers, for example, depending on whether they are integers, fractional numbers, etc. There is a type for Boolean values, specifically the values `True` and `False`. The types we are primarily concerned with in this chapter are `Char`, or "character," and `String`. A `String` is a list of characters.

It is easy to find out the type of a value, expression, or function in GHCi. We do this with the `:type` command.

Open up your REPL, enter `:type 'a'` at the prompt, and you should see something like this:

```
Prelude> :type 'a'
'a' :: Char
```

We'll highlight a few things here. First, we've enclosed our character in single quotes. This lets GHCi know that the character is not a variable. If you enter `:type a` instead, it will think it's a variable and give you an error message that the `a` is not in scope. That is, the variable `a` hasn't been defined (is not available in the current session), so it has no way to know what the type of `a` is.

Second, the `::` symbol is read as "has the type." You'll see this often in Haskell. Whenever you see that double colon, you know you're looking at a type signature. A type signature is a line of code that defines the types for a value, expression, or function.

And, finally, there is `Char`, the type. `Char` is the type that includes alphabetic characters, Unicode characters, symbols, etc. So, asking GHCi `:type 'a'`, that is, "what is the type of `a`?", gives us the information `'a' :: Char`, that is, "`a` has the type of `Char`."

Now, let's try a string of text. This time we have to use double quotation marks, not single, to tell GHCi we have a string, not a single character:

```
Prelude> :type "Hello!"
"Hello!" :: [Char]
```

We have something new in the type information. The square brackets around `Char` here are the syntactic sugar for a list. `String` is a *type alias*, or type synonym, for a list of `Char`. A type alias is what it sounds like: we use one name for a type, usually for convenience, that has a different type name underneath. Here, `String` is another name for a list of characters. By using the name `String`, we are able to visually differentiate it from other types of lists. When we talk about lists in more detail later, we'll see why the square brackets are considered syntactic sugar; for now, we only need to understand that GHCi says that `"Hello!"` has the type list of `Char`.

3.3 Printing simple strings

Now, let's see how to print strings of text in the REPL:

```
Prelude> print "hello world!"
"hello world!"
```

We use `print` to tell GHCi to print the string to the display, so it does, with the quotation marks still around it. The `print` function is not specific to strings of text, though; it can be used to print different types of data to the screen.

The following functions also tell GHCi to print to the screen but are specific to `String`:

```
Prelude> putStrLn "hello world!"
hello world!
Prelude>

Prelude> putStr "hello world!"
hello world!Prelude>
```

You can probably see that `putStr` and `putStrLn` are similar, with one key difference. We also notice that both of these print the string to the display without the quotation marks. This is because, while they are superficially similar to `print`, they have a different type than `print` does. Functions that are similar on the surface can behave differently depending on the type, or category, they belong to.

Next, let's take a look at how to do these things from source files. Type the following into a file named `print1.hs`:

```
-- print1.hs
module Print1 where

main :: IO ()
main = putStrLn "hello world!"
```

Here's what you should see when you load it in GHCi and run `main`:

```
Prelude> :l print1.hs
[1 of 1] Compiling Print1
Ok, one module loaded.
*Print1> main
hello world!
*Print1>
```

First, note that your `Prelude>` prompt may have changed to reflect the name of the module you loaded. You can use `:module` or `:m` to unload the module and return to `Prelude` if you wish. You can also set your prompt to something specific, which means it won't change every time you load or unload a module:[1]

```
Prelude> :set prompt "λ> "
λ> :r
Ok, one module loaded.
λ> main
hello world!
λ>
```

Looking at the code, `main` is the default action when you build an executable or run it in a REPL. It is not a function but is often a series of instructions to execute, which can include applying functions and producing side effects. When building a project with Stack, having a `main` executable in a `Main.hs` file is obligatory, but you can have source files and load them in GHCi without necessarily having a `main` block.

As you can see, `main` has the type `IO ()`. IO, or I/O, stands for input/output. In Haskell, it is a special type, called `IO`, used when the result of running a program involves effects beyond evaluating a function or expression. Printing to the screen is an effect, so printing the output of a module must be wrapped in this `IO` type. When you enter functions directly into the REPL, GHCi implicitly understands and implements `IO` without you having to specify that. Since the `main` action is the default executable, you will see it in a lot of source files that we build from here on out. We will explain its meaning in more detail in a later chapter.

Let's start another file:

[1] You can change it permanently, if you prefer, by setting the configuration in your `~/.ghci` file. You may have to create that file yourself, in order to do so.

```
-- print2.hs
module Print2 where

main :: IO ()
main = do
  putStrLn "Count to four for me:"
  putStr   "one, two"
  putStr   ", three, and"
  putStrLn " four!"
```

This `do` syntax is a special syntax that allows for sequencing actions. It is most commonly used to sequence the actions that constitute your program, some of which will necessarily perform effects such as printing to the screen (that's why the obligatory type of `main` is `IO ()`). `do` notation isn't strictly necessary, but since it often makes for more readable code than the alternatives, you'll see it a lot.

Here's what you should see when you run this one:

```
Prelude> :l print2.hs
[1 of 1] Compiling Print2
Ok, one module loaded.
Prelude> main
Count to four for me:
one, two, three, and four!
Prelude>
```

For a bit of fun, change the invocations of `putStr` to `putStrLn` and vice versa. Rerun the program, and see what happens. The `Ln` in `putStrLn` indicates that it starts a new line.

String concatenation

To *concatenate* means to *link together*. Usually, when we talk about concatenation in programming, we're talking about linear sequences such as lists or strings of text. If we concatenate the two strings `"Curry"` and `" Rocks!"`, we will get the string `"Curry Rocks!"`. Note the space at the beginning of `" Rocks!"`. Without that, we'd get `"CurryRocks!"`.

Let's start a new text file and type the following:

```haskell
-- print3.hs
module Print3 where

myGreeting :: String
myGreeting = "hello" ++ " world!"

hello :: String
hello = "hello"

world :: String
world = "world!"

main :: IO ()
main = do
  putStrLn myGreeting
  putStrLn secondGreeting
  where secondGreeting =
          concat [hello, " ", world]
```

We use `::` to declare the types of each top-level expression. It isn't strictly necessary, as the compiler can infer these types, but it is a good habit to be in as you write longer programs.

Remember, `String` is a type synonym for `[Char]`. You can try changing the type signatures to reflect that, and see if it changes anything in the program execution.

If you execute the code above, you should see something like this:

```
Prelude> :load print3.hs
[1 of 1] Compiling Print3
Ok, one module loaded.
*Print3> main
hello world!
hello world!
*Print3>
```

This little exercise demonstrates a few things:

1. We define values at the top level of a module: (`myGreeting`, `hello`, `world`, and `main`). That is, they are declared at the top level so that they are available throughout the module.

2. We specify explicit types for top-level definitions.
3. We concatenate strings with `++` and `concat`.

3.4 Top-level versus local definitions

What does it mean for something to be at the top level of a module? It doesn't necessarily mean it's defined at the top of the file. When the compiler reads the file, it will see all the top-level declarations, no matter in what order they come in the file (with some limitations, which we'll see later). Top-level declarations are not nested within anything else, and they are in scope throughout the whole module.

We can contrast a top-level definition with a local definition. To be locally defined would mean the declaration is nested within some other expression and is not visible outside that expression. We practiced this in the previous chapter with `let` and `where`. Here's an example for review:

```
module TopOrLocal where

topLevelFunction :: Integer -> Integer
topLevelFunction x =
  x + woot + topLevelValue
  where woot :: Integer
        woot = 10

topLevelValue :: Integer
topLevelValue = 5
```

In the code above, you could import and use `topLevelFunction` or `topLevelValue` from another module, and they are accessible to everything else in the module. However, `woot` is effectively invisible outside of `topLevelFunction`. The `where` and `let` clauses in Haskell introduce local bindings or declarations. To bind or declare something means to give an expression a name. You could pass around and use an anonymous version of `topLevelFunction` manually, but giving it a name and reusing it by that name is less repetitious.

Also note that we explicitly declare the type of `woot` in the `where` clause, using the `::` syntax. This isn't necessary (Haskell's type inference can figure it out), but it is done here to show you how. Be sure to load and run this code in your REPL:

```
Prelude> :l TopOrLocal.hs
[1 of 1] Compiling TopOrLocal
Ok, one module loaded.
*TopOrLocal> topLevelFunction 5
20
```

Experiment with different arguments and make sure you understand the results you're getting by walking through the arithmetic in your head (or on paper).

Exercises: Scope

1. These lines of code are from a REPL session. Is `y` in scope for `z`?

   ```
   Prelude> x = 5
   Prelude> y = 7
   Prelude> z = x * y
   ```

2. These lines of code are from a REPL session. Is `h` in scope for `g`? Go with your gut here:

   ```
   Prelude> f = 3
   Prelude> g = 6 * f + h
   ```

3. This code sample is from a source file. Is everything we need to execute `area` in scope?

   ```
   area d = pi * (r * r)
   r = d / 2
   ```

4. This code is also from a source file. Now, are `r` and `d` in scope for `area`?

   ```
   area d = pi * (r * r)
      where r = d / 2
   ```

3.5 Types of concatenation functions

Let's look at the types of `++` and `concat`. The `++` function is an infix operator. When we need to refer to an infix operator in a position that is not infix—such as when we are using it in a prefix position or having it stand alone in order to query its type—we must put parentheses around it. On the other hand, `concat` is a normal (not infix) function, so parentheses aren't necessary:

```
++     has the type [a]   -> [a] -> [a]
concat has the type [[a]] -> [a]
```

Here's how we query that in GHCi:

```
Prelude> :t (++)
(++) :: [a] -> [a] -> [a]
Prelude> :t concat
concat :: [[a]] -> [a]
```

If you are using GHC 7.10 or newer, and you probably are, `concat`'s type signature will look different. We'll explain it in detail later; for now, please read `Foldable t => t [a]` as being `[[a]]`. The `Foldable t` can be thought of as another list. List isn't the only possible type here—any type that has an instance of the `Foldable` type class would work—but right now, list is the only one we care about.

The type of `concat` says that it takes a list of lists as an input and returns a list. The new list will have the same values inside it as the list of lists; `concat` just "flattens" the original list into one list structure, in a manner of speaking. A `String` is a list, a list of `Char` specifically, and `concat` can work on lists of strings or lists of lists of other things:

```
Prelude> concat [[1, 2], [3, 4, 5], [6, 7]]
[1,2,3,4,5,6,7]
Prelude> concat ["Iowa", "Melman", "Django"]
"IowaMelmanDjango"
```

But what do these types mean? Here's how we can break them down:

```
(++) :: [a] -> [a] -> [a]
--      [1]    [2]    [3]
```

Everything after the `::` is about our types, not our values. The `a` inside the list type constructor `[]` is a type variable.

1. Take an argument of type `[a]`. This type is a list of elements of some type `a`. This function does not know what type `a` is. It doesn't need to know. In the context of the program, the type of `a` will be known and made concrete at some point.
2. Take another argument of type `[a]`, a list of elements whose type we don't know. Because the variables are the same, they must be the same type throughout (`a == a`).
3. Return a result of type `[a]`.

As we'll see, because `String` is a list containing characters, the operators we use with strings can also be used on lists of other types, such as lists of numbers. The type `[a]` means that we have a list with elements of a type `a` we do not yet know. If we use the operators to concatenate lists of numbers, then the `a` in `[a]` will be some type of number (for example, integers). If we are concatenating lists of characters, then `a` represents a `Char` because `String` is really type `[Char]`. The type variable `a` in `[a]` is polymorphic. *Polymorphism* is an important feature of Haskell. For concatenation, every list must be a list containing the same type of values; we cannot concatenate a list of numbers with a list of characters, for example. However, since the `a` is a variable at the type level, the literal values in each list we concatenate need not be the same, only the same *type*. In other words, `a` must equal `a`:

```
Prelude> "hello" ++ " Chris"
"hello Chris"
```

But:

```
Prelude> "hello" ++ [1, 2, 3]

• No instance for (Num Char) arising from
    the literal ‘1’
• In the expression: 1
  In the second argument of ‘(++)’,
    namely ‘[1, 2, 3]’
  In the expression: "hello" ++ [1, 2, 3]
```

In the first example, we have two strings, so the type of `a` matches—they're both `Char` in `[Char]`, even though the literal values are different. Since the type matches, no type error occurs, and we see the concatenated result.

In the second example, we have two lists (a `String` and a list of numbers) whose types do not match, so we get the error message. GHCi asks for an instance of the numeric type class `Num` for the type `Char`. *Type classes* provide definitions of operations, or functions, that can be shared across sets of types. For now, you can understand this message as telling you the types don't match, so it can't concatenate the two lists.

Exercises: Syntax errors

Read the syntax of the following functions, and decide whether it will compile. Test them in your REPL, and try to fix the syntax errors where they occur:

1. `++ [1, 2, 3] [4, 5, 6]`
2. `'<3' ++ ' Haskell'`
3. `concat ["<3", " Haskell"]`

3.6 Concatenation and scoping

We will use parentheses to call `(++)` as a prefix (not infix) function:

```
-- print3flipped.hs
module Print3Flipped where

myGreeting :: String
myGreeting = (++) "hello" " world!"

hello :: String
hello = "hello"

world :: String
world = "world!"
```

```
main :: IO ()
main = do
  putStrLn myGreeting
  putStrLn secondGreeting
  where secondGreeting =
          (++) hello ((++) " " world)
  -- could've been:
  --     secondGreeting =
  --       hello ++ " " ++ world
```

In `secondGreeting`, using `(++)` as a prefix function forces us to shift some things around. Parenthesizing it that way emphasizes its right associativity. Since it's an infix operator, you can check for yourself that it's right associative:

```
Prelude> :i (++)
(++) :: [a] -> [a] -> [a]
infixr 5 ++
```

The `where` clause creates local bindings for expressions that are not visible at the top level. In other words, the `where` clause in `main` introduces a definition visible only within the expression or function it's attached to, rather than making it visible to the entire module. Something visible at the top level is in scope for all parts of the module and may be exported by the module or imported by a different module. Local definitions, on the other hand, are only visible to that one function. You cannot import into a different module and reuse `secondGreeting`.

To illustrate:

```
-- print3broken.hs
module Print3Broken where

printSecond :: IO ()
printSecond = do
  putStrLn greeting
```

```
main :: IO ()
main = do
  putStrLn greeting
  printSecond
  where greeting = "Yarrrrr"
```

You should get an error like this:

```
Prelude> :l print3broken.hs
[1 of 1] Compiling Print3Broken

print3broken.hs:6:12: error:
    Variable not in scope:
      greeting :: String
  |
9 |   putStrLn greeting
  |            ^^^^^^^^
Failed, no modules loaded.
```

Let's take a closer look at this error:

```
print3broken.hs:6:12: error:
              [1][2]
    Variable not in scope:
                [  3  ]
      greeting :: String
      [  4  ]
```

1. The line the error occurs on: in this case, line 6.
2. The column the error occurs on: column 12. Text on computers is often described in terms of lines and columns. These line and column numbers are about lines and columns in your text file containing the source code.
3. The actual problem: we refer to something not in *scope*, that is, not *visible* to the `printSecond` function.
4. The thing we refer to that isn't visible, or in scope.

Now, make the `Print3Broken` code compile. It should print `"Yarrrrr"` twice on two different lines and then exit.

3.7 More list functions

Since a `String` is a list of `Char` values, you can use standard list operations on strings, as well.

Here are some examples:

```
Prelude> :t 'c'
'c' :: Char
Prelude> :t "c"
"c" :: [Char]
-- [Char] is String
```

The `:` operator, called *cons*, builds a list:

```
Prelude> 'c' : "hris"
"chris"
Prelude> 'P' : ""
"P"
```

Next up, `head` returns the head or first element of a list:

```
Prelude> head "Papuchon"
'P'
```

The complementary function `tail` returns the list with the head chopped off:

```
Prelude> tail "Papuchon"
"apuchon"
```

`take` returns the specified number of elements from the list, starting from the left:

```
Prelude> take 1 "Papuchon"
"P"
Prelude> take 0 "Papuchon"
""
Prelude> take 6 "Papuchon"
"Papuch"
```

And `drop` returns the remainder of the list after the specified number of elements have been dropped:

```
Prelude> drop 4 "Papuchon"
"chon"
Prelude> drop 9001 "Papuchon"
""
Prelude> drop 1 "Papuchon"
"apuchon"
```

We've already seen the `++` operator:

```
Prelude> "Papu" ++ "chon"
"Papuchon"
Prelude> "Papu" ++ ""
"Papu"
```

The infix operator, `!!`, returns the element that is in the specified position in the list. Note that this is an indexing function, and indices start from 0. That means the first element of your list is 0, not 1, when using this function:

```
Prelude> "Papuchon" !! 0
'P'
Prelude> "Papuchon" !! 4
'c'
```

Note also that while all of these are standard `Prelude` functions, many of them are considered *unsafe*. They are unsafe, because they do not know how to handle an empty list. Instead, they throw out an error message, or *exception*, when given an empty list as input:

```
Prelude> head ""
*** Exception: Prelude.head: empty list
Prelude> "" !! 4
*** Exception: Prelude.!!: index too large
```

This isn't ideal behavior, so the use of these functions is considered unwise for programs of any real size or complexity, although we will use them in these first few chapters. We will address how to cover all cases and make safer versions of such functions in a later chapter.

3.8 Chapter exercises

Reading syntax

1. For the following lines of code, read the syntax carefully, and decide whether they are written correctly. Test them in your REPL in order to check your work. Correct as many as you can:

   a) `concat [[1, 2, 3], [4, 5, 6]]`

   b) `++ [1, 2, 3] [4, 5, 6]`

   c) `(++) "hello" " world"`

   d) `["hello" ++ " world]`

   e) `4 !! "hello"`

   f) `(!!) "hello" 4`

   g) `take "4 lovely"`

   h) `take 3 "awesome"`

2. Next, we have two sets. The first set is lines of code, and the other is a set of results. Read the code, and figure out which results come from which lines of code. Be sure to test them in the REPL:

   a) `concat [[1 * 6], [2 * 6], [3 * 6]]`

   b) `"rain" ++ drop 2 "elbow"`

   c) `10 * head [1, 2, 3]`

   d) `(take 3 "Julie") ++ (tail "yes")`

   e)
   ```
   concat [tail [1, 2, 3],
           tail [4, 5, 6],
           tail [7, 8, 9]]
   ```

   Now, match each of the previous expressions to one of these results, which are presented in a scrambled order:

   a) `"Jules"`

   b) `[2,3,5,6,8,9]`

   c) `"rainbow"`

   d) `[6,12,18]`

   e) `10`

Building functions

1. Given the list-manipulation functions mentioned in this chapter, write functions that take the given inputs and return the expected outputs. Do them directly in your REPL, and use the `take` and `drop` functions you've already seen.

   Example

   If you apply your function to this value:

   ```
   "Hello World"
   ```

   Your function should return:

   ```
   "ello World"
   ```

   The following would be a fine solution:

   ```
   Prelude> drop 1 "Hello World"
   "ello World"
   ```

   Now, write expressions to perform the following transformations, just with the functions you've seen in this chapter. You do not need to do anything clever here:

   a)
   ```
   -- Given
   "Curry is awesome"
   -- Return
   "Curry is awesome!"
   ```

   b)
   ```
   -- Given
   "Curry is awesome!"
   -- Return
   "y"
   ```

   c)
   ```
   -- Given
   "Curry is awesome!"
   -- Return
   "awesome!"
   ```

2. Take each of the above, and rewrite it in a source file as a general function that could take different string inputs as arguments but retain the same behavior. Use a variable as an argument to each of your (named) functions. If you're unsure how to do this, refresh your memory by looking at the `waxOff` exercise from the previous chapter and the `TopOrLocal` module from this chapter.

3. Write a function of type `String -> Char` that returns the third character in a `String`. Remember to give the function a name, and apply it to a variable, not a specific `String`, so that it could be reused for different `String` inputs, as demonstrated. Feel free to name the function something else. Be sure to fill in both the type signature and the function definition after the equals sign:

```
thirdLetter ::
thirdLetter x =

-- If you apply your function
-- to this value:
"Curry is awesome"
-- Your function should return:
'r'
```

   Note that programming languages conventionally start indexing things by zero, so getting the "zeroth" index of a string will get you the first letter. Accordingly, indexing with 3 will get you the fourth. Keep this in mind as you write this function.

4. Change the above function so that the string operated on is always the same and the variable represents the number of the letter you want to return (you can use `"Curry is awesome!"` as your string input or a different string, if you prefer):

```
letterIndex  :: Int -> Char
letterIndex x =
```

5. Using the `take` and `drop` functions we looked at above, see if you can write a function called `rvrs` (an abbreviation of "reverse," used because there is already a function called `reverse` in Prelude, so if you give your function the same name, you'll get an error

message). `rvrs` should take the string `"Curry is awesome"` and return the result `"awesome is Curry"`. This may not be the most lovely Haskell code you will ever write, but it is entirely possible using only what we've learned so far. First, write it as a single function in a source file. This doesn't need to, and shouldn't, work for reversing the words of *any* sentence. You're expected only to slice and dice this particular string with `take` and `drop`.

6. Let's see if we can expand that function into a module. Why would we want to? By expanding it into a module, we can add more functions later that can interact with one another. We can also then export it to other modules if we want to and use this code in those other modules. There are different ways you could lay it out, but for the sake of convenience, we'll show you a sample layout so that you can fill in the blanks:

```
module Reverse where

rvrs :: String -> String
rvrs x =

main :: IO ()
main = print ()
```

Into the parentheses after `print`, you'll need to fill in your function name, `rvrs`, plus the argument you're applying `rvrs` to, in this case `"Curry is awesome"`. The `rvrs` function plus its argument is now the argument to `print`. It's important to put this function and argument combination inside the parentheses, so that the function gets applied and evaluated first, and then the result is printed.

Of course, we have also mentioned that you can use the `$` function to avoid using parentheses, too. Try modifying your main function to use that instead of parentheses.

3.9 Definitions

1. A *string* is a sequence of characters. In Haskell, `String` is represented by a linked list of `Char` values, aka `[Char]`.

2. A *type* or datatype is a classification of values or data. Types in Haskell determine what values are members of the type or that *inhabit* the type. Unlike in other languages, datatypes in Haskell by default do not delimit the operations that can be performed on that data.

3. *Concatenation* is the joining together of sequences of values. Often in Haskell, this is meant with respect to the `[]`, or list, datatype, which also applies to `String` (which, as we know, is `[Char]`). The *concatenation* function in Haskell is `++`, which has type `[a] -> [a] -> [a]`. For example:

   ```
   Prelude> "tacos" ++ " " ++ "rock"
   "tacos rock"
   ```

   A *scope* is where a variable can be validly referred to by name in a program. Something of global scope can be referred to anywhere in the entire program itself. A declaration at the top level of a Haskell module has module scope, and anything in that same module can refer to it. Another word used with the same meaning is *visibility*, because if a variable isn't *visible*, then it's not in *scope* and cannot be referenced by name.

4. *Local bindings* are bindings local to particular expressions. The primary distinction here from *top level* bindings is that *local* bindings cannot be imported by other programs or modules.

5. *Top level bindings* in Haskell are bindings that stand outside of any other declaration. The main feature of top-level bindings is that they can be made available to other modules within your programs or to other people's programs.

6. *Data structures* are a way of organizing data so that it can be accessed conveniently or efficiently.

Chapter 4

Basic Datatypes

> There are many ways of trying to understand programs. People often rely too much on one way, which is called "debugging" and consists of running a partly-understood program to see if it does what you expected. Another way, which ML advocates, is to install some means of understanding in the very programs themselves.
>
> Robin Milner

4.1 Basic datatypes

Haskell has a robust and expressive type system. Types play an important role in the readability, safety, and maintainability of Haskell code as they allow us to classify and delimit data, thus restricting the forms of data our programs can process. Types, also called *datatypes*, provide the means to build programs more quickly and also allow for greater ease of maintenance. As we learn more Haskell, we'll learn how to leverage types in a way that lets us accomplish the same things but with *less* code.

So far, we've looked at expressions involving numbers, characters, and lists of characters, also called strings. These are some of the standard datatypes and are built into the standard library. While those are useful datatypes and cover a lot of types of values, they don't cover every type of data. In this chapter, we will:

- Review types we have seen in previous chapters.
- Learn about datatypes, type constructors, and data constructors.
- Work with predefined datatypes.
- Learn more about type signatures and a bit about type classes.

4.2 What are types?

Expressions, when evaluated, reduce to values. Every value has a type. Types are how we group a set of values together that share something in common. Sometimes that commonality is abstract; sometimes it's a specific model of a particular concept or domain. If you've taken a mathematics course that covered sets, thinking about types as being like sets will help guide your intuition on what types are and how they work, in a mathematical sense.[1]

4.3 Anatomy of a data declaration

Data declarations are how datatypes are defined.

[1]Set theory is the study of mathematical collections of objects. Set theory was a precursor to type theory, the latter being used prolifically in the design of programming languages like Haskell. Logical operations like disjunction (or) and conjunction (and) used in the manipulation of sets have equivalents in Haskell's type system.

The type constructor is the name of the type and is capitalized. When you are reading or writing type signatures (the type level of your code), the type names or type constructors are what you use.

Data constructors are the values that inhabit the type they are defined in. They are the values that show up in your code, at the term level instead of the type level. By *term level*, we mean they are the values as they appear in your code or the values that your code evaluates to.

We will start with a basic datatype to see how datatypes are structured and get acquainted with the vocabulary. `Bool` isn't a datatype we've seen yet in this book, but it provides for truth values. It is named after the logician George Boole and the eponymous system of logic he devised. Because there are only two truth values, there are only two data constructors:

```
data Bool = False | True
--   [1]    [2] [3] [4]
```

1. Type constructor for datatype `Bool`. This is the name of the type and shows up in type signatures.
2. Data constructor for the value `False`.
3. The pipe `|` indicates a *sum type* or logical disjunction: *or*. So, a `Bool` value is `True` *or* `False`.
4. Data constructor for the value `True`.

This whole thing is called a data declaration. Data declarations do not always follow precisely the same pattern—there are datatypes that use logical conjunction (*and*) instead of disjunction, and some type constructors and data constructors may have arguments. The thing they have in common is the keyword `data` followed by the type constructor (or the name of the type that will appear in type signatures), the equals sign to denote a definition, and then data constructors (or the names of the values that inhabit term-level code).

You can find the datatype definition for built-in datatypes by using `:info` in GHCi:

```
Prelude> :info Bool
data Bool = False | True
```

Let's look at where different parts of datatypes show up in our code. If we query the type information for a function called `not`, we see that it takes one `Bool` value and returns another `Bool` value, so the type signature makes reference to the type constructor, or datatype name:

```
Prelude> :t not
not :: Bool -> Bool
```

But when we use the `not` function, we use the data constructors, or values:

```
Prelude> not True
False
```

And our expression evaluates to another data constructor, or value—in this case, the other data constructor for the same datatype.

Exercises: Mood swing

Given this datatype, answer the following questions:

```
data Mood = Blah | Woot deriving Show
```

The `deriving Show` part is not something we've explained yet. For now, all we'll say is that when you make your own datatypes, deriving `Show` allows the values of that type to be printed to the screen. We'll talk about it more when we talk about type classes in detail.

1. What is the type constructor, or name of this type?
2. If the function requires a `Mood` value, what are the values you could possibly use?
3. We are trying to write a function `changeMood` to change Chris's mood instantaneously. It should act like `not` in that, given one value, it returns the *other* value of the same type. So far, we've written a type signature `changeMood :: Mood -> Woot`. What's wrong with that?

4. Now we want to write the function that changes his mood. Given an input mood, it gives us the other one. Fix any mistakes and complete the function:

```
changeMood Mood = Woot
changeMood    _ = Blah
```

   We're doing something here called *pattern matching*. We can define a function by matching on a data constructor, or value, and describing the behavior that the function should have based on which value it matches. The underscore in the second line denotes a catch-all, "otherwise" case. So, in the first line of the function, we're telling it what to do in the case of a specific input. In the second one, we're telling it what to do regardless of *all potential inputs*. It's trivial when there are only two potential values of a given type, but as we deal with more complex cases, it can be convenient.

5. Enter all of the above—datatype (including the `deriving Show` bit), your corrected type signature, and the corrected function into a source file. Load it and run it in GHCi to make sure you got it right.

4.4 Numeric types

Let's look next at numeric types, because we've already seen these types in previous chapters, and numbers are familiar territory. It's important to understand that Haskell does not use only one type of number. For most purposes, the types of numbers we need to be concerned with are:

Integral numbers These are whole numbers, positive and negative.

1. `Int`: This type is a fixed-precision integer. By "fixed precision," we mean it has a range, with a maximum and a minimum, and so it cannot be arbitrarily large or small—more about this in a moment.

2. `Integer`: This type is also for integers, but this one supports arbitrarily large (or small) numbers.

3. `Word`: This is also a fixed-precision integer. Unlike `Int`, the smallest or lowest possible value expressible with a `Word` is the number zero. `Word` is suitable when you want to express whole digits that don't include negative numbers.

Fractional These are not integers. `Fractional` values include the following five types:

1. `Float`: This is the type used for single-precision floating point numbers. Fixed-point number representations have immutable numbers of digits assigned for the parts of the number before and after the decimal point. Sometimes fixed-point is called fixed-precision. In contrast, floating point can shift how many bits it uses to represent numbers before or after the decimal point. This flexibility does, however, mean that floating point arithmetic violates some common assumptions and should only be used with great care. Generally, floating point numbers should not be used at all in business applications.

2. `Double`: This is a double-precision floating point number type. It has twice as many bits with which to describe numbers as the `Float` type.

3. `Rational`: This is a fractional number that represents a ratio of two integers. The value `1 / 2 :: Rational` itself carries two `Integer` values, the numerator 1 and the denominator 2, and represents a ratio of 1 to 2. `Rational` is arbitrarily precise but not as efficient as `Scientific`.

4. `Fixed`: This is a fixed-point (or *fixed-precision*) type that can represent varying numbers of decimal points, depending on which type you choose. If your type is `Fixed E2`, then your values can track up to two digits after the decimal point. If the type is `Fixed E9`, then it could be up to nine digits after the decimal point. The `base` library provides `E0`, `E1`, `E2`, `E3`, `E6`, `E9`, and `E12`. It isn't too much trouble to add your own resolution if you require a different amount of precision.

5. `Scientific`: This is a space efficient and almost arbitrary precision scientific number type. `Scientific` numbers are represented

> using scientific notation. It stores the coefficient as an `Integer` and the exponent as an `Int`. Since `Int` isn't arbitrarily large, there is technically a limit to the size of number you can express with `Scientific`, but hitting that limit is unlikely. `Scientific` is available in a library[2] and can be installed using `cabal install` or `stack install`.

These numeric datatypes all have instances of a type class called `Num`. We will talk about type classes in upcoming chapters, but as we look at the types in this section, you will see `Num` listed in some of the type information.

Type classes are a way of adding functionality to types that is reusable across all the types that have instances of that type class. `Num` is a type class for which most numeric types will have an instance, because there are standard functions that are convenient to have available for all types of numbers. The `Num` type class is what provides your standard `+`, `-`, and `*` operators, along with a few others. Any type that has an instance of `Num` can be used with those functions. An instance defines how the functions work for that specific type. We will talk about type classes in much more detail soon.

Integral numbers

As we noted above, there are three main types of integral numbers: `Int`, `Integer`, and `Word`.

Integral numbers are whole numbers with no fractional component. The following are integral numbers:

```
1 2 199 32442353464675685678
```

The following are not integral numbers:

```
1.3 1/2
```

Integer

The numbers of type `Integer` are the sorts of numbers we're used to working with in arithmetic equations that involve whole numbers.

[2] Hackage page for `scientific`: `https://hackage.haskell.org/package/scientific`.

They can be positive or negative, and `Integer` extends as far in either direction as one needs them to go.

The `Bool` datatype only has two possible values, so we can list them explicitly as data constructors. In the case of `Integer`, and most numeric datatypes, these data constructors are not written out, because they include an infinite series of whole numbers. We'd need infinite data constructors stretching up and down from zero. Hypothetically, we could represent `Integer` as a sum of three cases: recursive constructors headed toward negative infinity, zero, and recursive constructors headed toward positive infinity. This representation would be terribly inefficient, so there's some GHC magic sprinkled on numbers, instead.

Why do we have Int?

The `Int` numeric types are artifacts of what computer hardware has supported natively over the years. Most programs should use `Integer`, not `Int`, unless the limitations of the type are understood and the additional performance makes a difference.

The danger of `Int` and the related types `Int8`, `Int16`, et al. is that they cannot express arbitrarily large quantities of information. Since they are integral, this means they cannot be arbitrarily large in the positive or negative sense.

Here's what happens if we try to represent a number too large for `Int8`:

```
Prelude> import GHC.Int
Prelude> 127 :: Int8
127
Prelude> 128 :: Int8

Literal 128 is out of the
  Int8 range -128..127
If you are trying to write a large negative
  literal, use NegativeLiterals
-128
Prelude> (127 + 1) :: Int8
-128
```

The syntax you see above, the type annotation `:: Int8`, assigns the `Int8` type to the expression `127 + 1`. As we will explain in more detail in the next chapter, numbers are polymorphic under the surface, and the compiler doesn't assign them a concrete type until it is forced to. It would be weird and unexpected if the compiler defaulted all numbers to `Int8`, so in order to reproduce the situation of having a number too large for an `Int` type, we have to assign it that concrete type ourselves.

As you can see, 127 is fine, because it is within the range of valid values of type `Int8`, but 128 gives you a warning about the impending overflow, and 127 + 1 overflows the bounds and resets back to its smallest numeric value. Because the memory the value is allowed to occupy is fixed for `Int8`, it cannot grow to accommodate the value 128 the way `Integer` can. Here, the 8 represents how many bits the type uses to represent integral numbers.[3] Being of a fixed size can be useful in some applications, but most of the time, `Integer` is preferred.

You can find out the minimum and maximum bounds of numeric types using `maxBound` and `minBound` from the `Bounded` type class. Here's an example using `Int8` and `Int16`:

```
Prelude> import GHC.Int
Prelude> :t minBound
minBound :: Bounded a => a
Prelude> :t maxBound
maxBound :: Bounded a => a

Prelude> minBound :: Int8
-128
Prelude> minBound :: Int16
-32768
Prelude> minBound :: Int32
-2147483648
Prelude> minBound :: Int64
-9223372036854775808

Prelude> maxBound :: Int8
127
```

[3]The representation used for the fixed-size `Int` types is *two's complement*.

```
Prelude> maxBound :: Int16
32767
Prelude> maxBound :: Int32
2147483647
Prelude> maxBound :: Int64
9223372036854775807
```

Thus, you can find the limitations of possible values for any type that has an instance of that particular type class. In this case, we are able to find out that the range of values we can represent with an `Int8` is -128 to 127.

You can find out if a type has an instance of `Bounded`, or any other type class, by asking GHCi for the `:info` for that type. Doing this will also give you the datatype representation for the type you query:

```
Prelude> :i Int
data Int = I# GHC.Prim.Int#
instance Bounded Int
```

`Int`, of course, has many more type class instances, but `Bounded` is the one we care about at this time.

Word

The purpose and reasons for using `Word` are similar to that of `Int`, it just expresses a different numeric range that doesn't include negative numbers. A demonstration:

```
Prelude> import Data.Word
Prelude> minBound :: Word8
0
Prelude> maxBound :: Word8
255
Prelude> minBound :: Word16
0
Prelude> maxBound :: Word16
65535
```

You might've noticed that the upper bounds of `Word` types are approximately double that of the same size `Int` types. That's because

the bit in `Int` values used to express negative numbers is now used to signify extra positive numbers. Follow our reference to *two's complement* earlier in this chapter to understand how this works.

Fractional numbers

Five common `Fractional` types used in Haskell are `Float`, `Double`, `Rational`, `Fixed`, and `Scientific`. `Rational`, `Double`, `Fixed`, and `Float` come with your install of GHC. `Scientific` comes from a library, as we mentioned previously. `Rational` and `Scientific` are arbitrary precision, with the latter being more efficient. Arbitrary precision means that these can be used to do calculations requiring a high degree of precision rather than being limited to a specific degree of precision, the way `Float` and `Double` are. You almost never want a `Float`, unless you're doing graphics programming such as with OpenGL.

Some computations involving numbers will be fractional rather than integral. A good example of this is the division function, `/`, which has the type:

```
Prelude> :t (/)
(/) :: Fractional a => a -> a -> a
```

The notation `Fractional a =>` denotes a *type class constraint*. It tells us that the type variable `a` must implement the `Fractional` type class. Whatever type of number `a` turns out to be, it must be a type that has an instance of the `Fractional` type class; that is, there must be a declaration of how the operations from that type class will work for that particular type. The `/` function will take one number that implements `Fractional`, divide it by another of the same type, and return a value of the same type as the result.

`Fractional` is a type class that requires types to already have an instance of the `Num` type class. We describe this relationship between type classes by saying that `Num` is a superclass of `Fractional`. So `+` and other functions from the `Num` type class can be used with `Fractional` numbers, but functions from the `Fractional` type class cannot be used with all types that have a `Num` instance:

Here's what happens when we use `/` in the REPL:

```
Prelude> 1 / 2
0.5
```

```
Prelude> 4 / 2
2.0
```

Note that even when we have a whole number as a result, the result is fractional. This is because values of `Fractional a => a` default to the floating point type `Double`. In most cases, you won't want to explicitly use `Double`. You may be better off using a fixed precision type such as the types in `Data.Fixed`. If you are tracking money, you definitely want fixed precision. Most people do not find it easy to reason about floating point arithmetic and find it difficult to code around the quirks. Those quirks exist by design, but that's another story. To avoid making mistakes, use fixed-precision types as a matter of course.

4.5 Comparing values

Up to this point, most of our operations with numbers have only involved simple arithmetic. We can also compare numbers to determine whether they are equal, greater than, or less than:

```
Prelude> x = 5
Prelude> x == 5
True
Prelude> x == 6
False
Prelude> x < 7
True
Prelude> x > 3
True
Prelude> x /= 5
False
```

Notice here that we first declare a value for `x` using the standard equals sign. Now we know that for the remainder of our REPL session, all instances of `x` will be the value *5*. Because the equals sign in Haskell is already used to define variables and functions, we must use a double equals sign, `==`, to have the specific meaning *is equal to*. The `/=` symbol means *is not equal to*. The other symbols should already be familiar to you.

Having done this, we see that GHCi is returning a result of either `True` or `False`, depending on whether the expression is true or false. `True` and `False` are the data constructors for the `Bool` datatype we saw above. If you look at the type information for any of these infix operators, you'll find the result type listed as `Bool`:

```
Prelude> :t (==)
(==) :: Eq a => a -> a -> Bool
Prelude> :t (<)
(<) :: Ord a => a -> a -> Bool
```

Notice that we get some type class constraints again. `Eq` is a type class that includes everything that can be compared and determined to be equal in value; `Ord` is a type class that includes all things that can be ordered. Note that neither of these is limited to numbers. Numbers can be compared and ordered, of course, but so can letters, so this type class constraint allows for flexibility. These equality and comparison functions can take any values that can be said to be equal in value or can be ordered. The rest of the type information tells us that it takes one of these values, compares it to another value of the same type, and returns a value of type `Bool`. As we've already seen, the `Bool` values are `True` or `False`.

With this information, let's try playing with some other values:

```
Prelude> 'a' == 'a'
True
Prelude> 'a' == 'b'
False
Prelude> 'a' < 'b'
True
Prelude> 'a' > 'b'
False
Prelude> 'a' == 'A'
False
Prelude> "Julie" == "Chris"
False
```

We know that alphabetical characters can be ordered, although we do not normally think of "a" as being less than "b." But we can

understand that here it means `'a'` comes before `'b'` in alphabetical order. Further, we see this also works with strings such as `"Julie"` or `"Chris"`. GHCi has faithfully determined that those two strings are not equal in value.

Now use your REPL to determine whether `'a'` or `'A'` is greater.

Next, take a look at this sample, and see if you can figure out why GHCi returns the given results:

```
Prelude> "Julie" > "Chris"
True
Prelude> "Chris" > "Julie"
False
```

Good to see Haskell code that reflects reality. `"Julie"` is greater than `"Chris"`, because `'J' > 'C'`. If the words had been `"Back"` and `"Brack"`, then it would've skipped the first letter to determine which is greater, because `'B' == 'B'`, and then we would find that `"Brack"` is greater, because `'r' > 'a'` in the lexicographic ordering. Note that this is leaning on the `Ord` type class instances for the list type *and* `Char`. You can only compare lists of items where the items themselves also have an instance of `Ord`. Accordingly, the following will work, because `Char` and `Integer` have instances of `Ord`:

```
Prelude> ['a', 'b'] > ['b', 'a']
False
Prelude> 1 > 2
False
Prelude> [1, 2] > [2, 1]
False
```

A datatype that has no instance of `Ord` will not work with these functions:

```
Prelude> data Mood = G | B deriving Show
Prelude> [G, B]
[G,B]
Prelude> [G, B] > [B, G]

• No instance for (Ord Mood) arising from a
    use of ‘>’
```

```
• In the expression: [G, B] > [B, G]
  In an equation for ‘it’:
    it = [G, B] > [B, G]
```

"No instance for `(Ord Mood)`" means that `Mood` doesn't have an `Ord` instance, so the type checker doesn't know how to order values of that type.

Here is something else that doesn't work with these functions:

```
Prelude> "Julie" == 8

• No instance for (Num [Char]) arising from
    the literal ‘8’
• In the second argument of ‘(==)’,
    namely ‘8’
  In the expression: "Julie" == 8
  In an equation for ‘it’:
    it = "Julie" == 8
```

We said above that comparison functions are polymorphic and can work with a lot of different types. But we also noted that the type information only admits values of matching types. Once you've entered in a term-level value that's a `String`, such as `"Julie"`, the type is determined, and the other argument must have the same type. Since numeric values are polymorphic, the type checker tries to reconcile the literal value `8` with a value of type `String`. In order to convert the `8` into a `String`, there must be an instance of `Num` for the type `[Char]`. No such instance exists, and therefore we receive the type error above.

4.6 Go on and Bool me

The `Bool` datatype comes standard in the `Prelude`. As we saw earlier, `Bool` is a sum type with two constructors:

```
data Bool = False | True
```

This declaration creates a datatype with the type constructor `Bool`. We refer to specific types by their type constructors, and we only use type constructors in type signatures, not in the expressions that make

up our term-level code. The type constructor `Bool` takes no arguments (some type constructors do take arguments). The definition of `Bool` above also creates two data constructors, `True` and `False`. Both of these values are of type `Bool`. Any function that accepts values of type `Bool` must allow for the possibility of `True` *or* `False`; you cannot specify in the type that it should only accept one specific value. An English language formulation of this datatype would be something like: "The datatype `Bool` is represented by the value `True` or the value `False`."

Remember, you can find the type of any value by asking for it in GHCi, just as you can with functions:

```
Prelude> :t True
True :: Bool
Prelude> :t "Julie"
"Julie" :: [Char]
```

Now, let's have some fun with `Bool`. We'll start by reviewing the `not` function:

```
Prelude> :t not
not :: Bool -> Bool
Prelude> not True
False
```

Note that we capitalize `True` and `False`, because they are our data constructors. What happens if you try to use `not` without capitalizing them?

Let's try something slightly more complex:

```
Prelude> x = 5
Prelude> not (x == 5)
False
Prelude> not (x > 7)
True
```

We know that comparison functions evaluate to a `Bool` value, so we can use them with `not`.

Let's play with infix operators that deal directly with Boolean logic. How do we use `Bool` and these associated functions?

First, `&&` is the infix operator for Boolean conjunction. The first example reads, colloquially, "true and true":

```
Prelude> True && True
True
Prelude> (8 > 4) && (4 > 5)
False
Prelude> not (True && True)
False
```

The infix operator for Boolean disjunction is `||`, so the first example here reads "false or true":

```
Prelude> False || True
True
Prelude> (8 > 4) || (4 > 5)
True
Prelude> not ((8 > 4) || (4 > 5))
False
```

We can look up info about datatypes that are in scope (if they're not in scope, we have to import the module they live in to bring them into scope) using the `:info` command GHCi provides. Scope is a way to refer to where a named binding to an expression is valid. When we say something is *in scope*, it means you can use that expression by its bound name, either because it was defined in the same function or module, or because you imported it. So, it's visible to the program we're trying to run right now. What is built into Haskell's `Prelude` module is significant, because everything in it is automatically imported and in scope. For now, this is what we want, so we don't have to write every function from scratch.

Exercises: Find the mistakes

The following lines of code may have mistakes—some of them won't compile! You know what you need to do:

1. `not True && true`
2. `not (x = 6)`
3. `(1 * 2) > 5`
4. `[Merry] > [Happy]`
5. `[1, 2, 3] ++ "look at me!"`

Conditionals with if-then-else

Haskell doesn't have "if" statements, but it does have *if expressions*. It's a built-in bit of syntax that works with the `Bool` datatype:

```
Prelude> t = "Truthin'"
Prelude> f = "Falsin'"
Prelude> if True then t else f
"Truthin'"
```

The expression `if True` evaluates to `True`, hence we return `t`:

```
Prelude> if False then t else f
"Falsin'"
Prelude> :t if True then t else f
if True then t else f :: [Char]
```

And `if False` evaluates to `False`, so we return the `else` value. The type of the whole expression is `String` (aka `[Char]`), because that's the type of the value that is returned as a result.

The structure here is:

```
if CONDITION
then EXPRESSION_A
else EXPRESSION_B
```

If the condition (which must evaluate to `Bool`) reduces to the value `True`, then `EXPRESSION_A` is the result, otherwise it's `EXPRESSION_B`.

An if expression can be thought of as a way to choose between two values. You can embed a variety of expressions within the `if` condition of an if-then-else, as long as it evaluates to `Bool`. The types of the expressions in the `then` and `else` clauses must be the same, as in the following:

```
Prelude> x = 0
Prelude> a = "AWESOME"
Prelude> w = "wut"
Prelude> if (x + 1 == 1) then a else w
"AWESOME"
```

Here's how it reduces:

```
-- Given:
x = 0

if (x + 1 == 1) then "AWESOME" else "wut"
-- x is zero

if (0 + 1 == 1) then "AWESOME" else "wut"
-- Reduce 0 + 1 so we can see
-- if it's equal to 1

if (1 == 1) then "AWESOME" else "wut"
-- Does 1 equal 1?

if True then "AWESOME" else "wut"
-- Pick the branch based on the Bool value

"AWESOME"
-- Dunzo.
```

But this does not work:

```
Prelude> dog = "adopt a dog"
Prelude> cat = "or a cat"
Prelude> x = 0
Prelude> if x * 100 then dog else cat

• No instance for (Num Bool) arising from a
  use of ‘*’
• In the expression: x * 100
  In the expression: if x * 100 then dog
    else cat
  In an equation for ‘it’: it = if x * 100
    then dog else cat
```

We get this type error, because the condition passed to the if-expression is of type `Num a => a`, not `Bool`, and `Bool` doesn't implement the `Num` type class. To oversimplify, `x * 100` evaluates to a numeric result, and numbers aren't truth values. It would have type checked had the condition been `x * 100 == 0` or `x * 100 == 9001`. In those cases, it would have been an equality check of two numbers that reduces to a `Bool` value.

Here's an example of a function that uses a `Bool` value in an if expression:

```
-- greetIfCool1.hs
module GreetIfCool1 where

greetIfCool :: String -> IO ()
greetIfCool coolness =
  if cool
    then putStrLn "eyyyyy. What's shakin'?"
  else
    putStrLn "pshhhh."
  where cool =
          coolness == "downright frosty yo"
```

When you test this in the REPL, it should play out like this:

```
Prelude> :l greetIfCool1.hs
[1 of 1] Compiling GreetIfCool1
Ok, one module loaded.
Prelude> greetIfCool "downright frosty yo"
eyyyyy. What's shakin'?
Prelude> greetIfCool "please love me"
pshhhh.
```

Also, note that `greetIfCool` could have been written with `cool` as a function—rather than as a value—defined against the argument directly, like so:

```
-- greetIfCool2.hs
module GreetIfCool2 where

greetIfCool :: String -> IO ()
greetIfCool coolness =
  if cool coolness
    then putStrLn "eyyyyy. What's shakin'?"
  else
    putStrLn "pshhhh."
  where cool v =
          v == "downright frosty yo"
```

4.7 Tuples

A *tuple* is a type that allows you to store and pass around multiple values within a single, composite value. Tuples have a distinctive, built-in syntax that is used at both type and term levels, and each tuple has a fixed number of constituents. We refer to tuples by the number of values in each tuple: the 2-tuple or pair, for example, has two values inside it, `(x, y)`; the 3-tuple or triple has three, `(x, y, z)`, and so on. This number is also known as the tuple's *arity*. As we will see, the values within a tuple do not have to be of the same type.

We will start by looking at the 2-tuple, a tuple with two elements. The 2-tuple is expressed at both the type level and the term level with the constructor `(,)`. The datatype declaration looks like this:

```
Prelude> :info (,)
data (,) a b = (,) a b
```

This is different from the `Bool` type we looked at earlier in a couple of important ways, even apart from that special type constructor syntax. The first is that it has two parameters, represented by the type variables `a` and `b`. Those have to be applied to concrete types, much as variables at the term level have to be applied to values in order to evaluate a function. The second major difference is that this is a *product type*, not a sum type. A product type represents a logical conjunction: you must supply *both* arguments to construct a value.

Notice that the two type variables are different, so that allows for tuples that contain values of two different types. The types are not, however, *required* to be different:

```
Prelude> (,) 8 10
(8,10)
Prelude> (,) 8 "Julie"
(8,"Julie")
Prelude> (,) True 'c'
(True,'c')
```

But if we try to apply it to only one argument:

```
Prelude> (,) 9
```

```
• No instance for
    (Show (b0 -> (Integer, b0)))
      arising from a use of ‘print’
      (maybe you haven't applied a function
      to enough arguments?)
• In a stmt of an interactive GHCi command:
    print it
```

Well, look at that error. This is one we will explore in more detail soon, but for now, the important part is the part in parentheses: we haven't applied the function—in this case, the data constructor—to enough arguments.

The 2-tuple in Haskell has some default convenience functions for extracting the first or second value. They're named `fst` and `snd`:

```
fst :: (a, b) -> a
snd :: (a, b) -> b
```

The type signature tells us there's nothing those functions could do other than return the first or second value, respectively.

Here are some examples of manipulating tuples, specifically the 2-tuple:

```
Prelude> myTup = (1 :: Integer, "blah")
Prelude> :t myTup
myTup :: (Integer, [Char])
Prelude> fst myTup
1
Prelude> snd myTup
"blah"
Prelude> import Data.Tuple
Prelude> swap myTup
("blah",1)
```

We have to import `Data.Tuple`, because `swap` isn't included in the `Prelude`.

We can also combine tuples with other expressions:

```
Prelude> 2 + fst (1, 2)
3
```

```
Prelude> 2 + snd (1, 2)
4
```

The `(x, y)` syntax of the tuple is special. The constructors you use in the type signatures and in your code (terms) are syntactically identical even though they're different things. Sometimes, that type constructor is referred to without the type variables explicitly inside of it, such as `(,)`. Other times, you'll see `(a, b)`—particularly in type signatures.

You can use that syntax to *pattern match* when you write functions, too. One nice thing about that is that the function definition can often look very much like the type signature. For example, we can implement `fst` and `snd` for ourselves, like this:

```
fst' :: (a, b) -> a
fst' (a, b) = a

snd' :: (a, b) -> b
snd' (a, b) = b
```

Let's look at another example of pattern matching on tuples:

```
tupFunc :: (Int, [a])
        -> (Int, [a])
        -> (Int, [a])
tupFunc (a, b) (c, d) =
  ((a + c), (b ++ d))
```

It's generally unwise to use tuples of an overly large size, both for efficiency and sanity reasons. Most tuples you see will be `( , , , , )` (5-tuple) or smaller.

4.8 Lists

Lists are another type used to contain multiple values within a single value. However, they differ from tuples in three important ways: First, all the elements of a list must be of the same type. Second, lists have their own distinct `[]` syntax. Like the tuple syntax, it is used for both the type constructor in type signatures and at the term level to express list values. Third, the number of values that will be in the list

isn't specified in the type—unlike tuples, where the arity is set in the type and immutable.

Here's an example for your REPL:

```
Prelude> p = "Papuchon"
Prelude> awesome = [p, "curry", ":)"]
Prelude> awesome
["Papuchon","curry",":)"]

Prelude> :t awesome
awesome :: [[Char]]
```

The first thing to note is that `awesome` is a list of lists of `Char` values, because it is a list of strings, and `String` is a type alias for `[Char]`. This means all the functions and operations valid for lists of any value, usually expressed as `[a]`, are valid for `String`, because `[Char]` is more specific than `[a]`:

```
Prelude> s = "The Simons"
Prelude> also = ["Quake", s]
Prelude> :t (++)
(++) :: [a] -> [a] -> [a]
Prelude> awesome ++ also
["Papuchon",
 "curry",
 ":)",
 "Quake",
 "The Simons"]

Prelude> allAwesome = [awesome, also]
Prelude> allAwesome
[["Papuchon","curry",":)"],
 ["Quake","The Simons"]]
Prelude> :t allAwesome
allAwesome :: [[[Char]]]
Prelude> :t concat
concat :: [[a]] -> [a]
Prelude> concat allAwesome
["Papuchon",
```

```
 "curry",
 ":)",
 "Quake",
 "The Simons"]
```

We'll save a full exploration of the list datatype until we get to the chapter on lists. The list data structure gets a whole chapter, because lists have some interesting complexity, we're going to use them to demonstrate some things about Haskell's non-strict evaluation, and there are *many* standard functions and constructs that can be used with lists.

4.9 Chapter exercises

As in previous chapters, you will gain more by working out the answers to the following exercises before using GHCi, but be sure to use your REPL to check them. Also, you will need to have the `awesome`, `also`, and `allAwesome` code from above in scope for this REPL session. For your convenience, here are those values again:

```
awesome = ["Papuchon", "curry", ":)"]
also = ["Quake", "The Simons"]
allAwesome = [awesome, also]
```

`length` is a function that takes a list and returns the number of items in that list.

1. Given the definition of `length` above, what would its type signature be? How many arguments does it take and of what types? What is the type of the result it evaluates to?

2. What are the results of the following expressions?

   a) `length [1, 2, 3, 4, 5]`

   b) `length [(1, 2), (2, 3), (3, 4)]`

   c) `length allAwesome`

   d) `length (concat allAwesome)`

3. Given what we know about numeric types and the type signature of `length`, look at the two expressions below. One works, and one returns an error. Determine which will return an error and why (n.b., you will find that `Foldable t => t a` represents `[a]`, as with `concat` in the previous chapter. Again, consider `Foldable t` to represent a list here, even though list is only one of the possible types of `t`):

```
6 / 3
-- and
6 / length [1, 2, 3]
```

4. How can you fix the broken code from the preceding exercise using a different division function/operator?
5. What is the type of the expression `2 + 3 == 5`? What would we expect as a result?
6. What is the type and expected result value of the following:
   a) `x = 5`
   b) `x + 3 == 5`
7. Below are some bits of code. Which will work? Why or why not? If they will work, what values would these reduce to?
   a) `length allAwesome == 2`
   b) `length [1, 'a', 3, 'b']`
   c) `length allAwesome + length awesome`
   d) `(8 == 8) && ('b' < 'a')`
   e) `(8 == 8) && 9`
8. Write a function that tells you whether or not a given `String` (or list) is a palindrome. Here, you'll want to use a function called `reverse`, a predefined function that does what it sounds like:

```
reverse :: [a] -> [a]
reverse "blah"
"halb"
```

```
isPalindrome :: (Eq a) => [a] -> Bool
isPalindrome x = undefined
```

9. Write a function to return the absolute value of a number using an if-then-else expression:

```
myAbs :: Integer -> Integer
myAbs = undefined
```

10. Fill in the definition of the following function, using `fst` and `snd`:

```
f :: (a, b) -> (c, d) -> ((b, d), (a, c))
f = undefined
```

Correcting syntax

In the following examples, you'll be shown syntactically incorrect code. Type it in, and try to correct it in your text editor, validating it with GHC or GHCi.

1. Here, we want a function that adds 1 to the length of a string argument and returns that result:

```
x = (+)

F xs = w 'x' 1
    where w = length xs
```

2. This is supposed to be the identity function, `id`:

```
\X = x
```

3. When fixed, this function will return 1 from the value `(1, 2)`:

```
f (a b) = A
```

Match the function names to their types

1. Which of the following types is the type of `show`?

   a) `show a => a -> String`

   b) `Show a -> a -> String`

c) `Show a => a -> String`

2. Which of the following types is the type of `==`?

   a) `a -> a -> Bool`

   b) `Eq a => a -> a -> Bool`

   c) `Eq a -> a -> a -> Bool`

   d) `Eq a => A -> Bool`

3. Which of the following types is the type of `fst`?

   a) `(a, b) -> a`

   b) `b -> a`

   c) `(a, b) -> b`

4. Which of the following types is the type of `+`?

   a) `Num a -> a -> a -> Bool`

   b) `Num a => a -> a -> Bool`

   c) `num a => a -> a -> a`

   d) `Num a => a -> a -> a`

   e) `a -> a -> a`

4.10 Definitions

1. A *tuple* is an ordered grouping of values. In Haskell, you cannot have a tuple with only one element, but there is a "zero" tuple, also called *unit* or `()`. The types of the elements of tuples are allowed to vary, so both `(String, String)` and `(Integer, String)` are valid tuple types. Tuples in Haskell are the usual means of briefly carrying around multiple values without giving that combination its own name.

2. A *type class* is a set of operations defined with respect to a polymorphic type. When a type has an instance of a type class, values of that type can be used in the standard operations defined for that type class. In Haskell, type classes are unique pairings of a class and a concrete instance. This means that if a given type `a` has an instance of `Eq`, it has *only one* instance of `Eq`.

3. *Data constructors* in Haskell provide a means of creating values that inhabit a given type. Data constructors in Haskell have a type and can either be constant values (nullary) or take one or more arguments, like functions. In the following example, `Cat` is a nullary data constructor for `Pet`, and `Dog` is a data constructor that takes an argument:

```
-- Why name a cat?
-- They don't answer anyway.
type Name = String

data Pet = Cat | Dog Name
```

The data constructors have the following types:

```
Prelude> :t Cat
Cat :: Pet
Prelude> :t Dog
Dog :: Name -> Pet
```

4. *Type constructors* in Haskell are *not* values and can only be used in type signatures. Just as data declarations generate data constructors in order to create the values that inhabit a given type, data declarations also generate *type constructors*, which can be used to denote that type. In the above example, `Pet` is the type constructor. A guideline for differentiating the two kinds of constructors is that type constructors always go to the left of the `=` in a data declaration.

5. *Data declarations* define new datatypes in Haskell. Data declarations *always* create a new type constructor but may or *may not* create new data constructors. Data declarations are how we refer to the entire definition that begins with the `data` keyword.

6. A *type alias* is a way to refer to a type constructor or type constant by an alternate name, usually to communicate something more specific or for brevity:

```
type Name = String
```

This creates a new type alias called `Name` of the type `String`, so this is *not* a data declaration, it's just a type alias declaration.

7. *Arity* is the number of arguments a function accepts. This notion is a little slippery in Haskell since, due to currying, all functions are arity one (or "unary"), and we handle accepting multiple arguments by nesting functions.

8. *Polymorphism* means being able to write code in terms of values that may be one of several types or of any type. Polymorphism in Haskell, specifically, is either *parametric* or *constrained*. The identity function, `id`, is an example of a parametrically polymorphic function:

```
id :: a -> a
id x = x
```

Here, `id` works for a value of *any* type, because it doesn't use any information specific to a given type or set of types. Whereas, the following function, `isEqual`, is polymorphic, but *constrained* or *bounded* to the set of types that have an instance of the `Eq` type class:

```
isEqual :: Eq a => a -> a -> Bool
isEqual x y = x == y
```

The different kinds of polymorphism will be discussed in greater detail in a later chapter.

4.11 Names and variables

Names

In Haskell, there are seven categories of entities that have names: functions, term-level variables, data constructors, type variables, type constructors, type classes, and modules. Term-level variables and data constructors exist in your terms. *Term level* is where your values live and is the code that executes when your program is running. At the *type level*, which is used during the static analysis and verification of your program, we have type variables, type constructors, and type classes. Lastly, for the purpose of organizing our code into coherent groupings across different files, we have modules.

Conventions for variables

Haskell uses a lot of variables, and some conventions have developed for naming them. It's not critical that you memorize these conventions, however, because for the most part, they are merely common practices. But familiarizing yourself with them will help you read Haskell code written by others.

Type variables (that is, variables in type signatures) generally start at `a` and go from there: `a`, `b`, `c`, and so forth. You may sometimes see them with numbers appended to them, e.g., `a1`.

Functions can be used as arguments and in that case are typically labeled with variables starting at `f` (followed by `g` and so on). They may sometimes have numbers appended (e.g., `f1`) and may also sometimes be decorated with the `'` character, as in `f'`. This would be pronounced "eff-prime," should you need to say it aloud. Usually, this would denote a function that is closely related to, or a helper function of, a function `f`. Functions may also be given variable names that are not on this spectrum as a mnemonic. For example, a function that results in a list of prime numbers might be called `p`, while a function that fetches some text might be called `txt`.

Variables do not have to be a single letter. In small programs, they often are; in larger programs, they usually should not be a single letter. If there are many variables in a function or program, as is common, it is helpful to have descriptive variable names. It is often advisable in domain-specific code to use domain-specific variable names.

Arguments to functions are most often given names starting at `x`, again occasionally seen numbered as in `x1`. Other single-letter variable names may be chosen when they serve a mnemonic role, such as choosing `r` to represent a value that is the radius of a circle.

If you have a list of items, and you have used the name `x` to refer to one such item, by convention the list itself will usually be called `xs`, that is, the plural of `x`. You will often see this convention in the form `(x:xs)`, which means you have a list where the "head," or first element, is `x`, and the rest of the list, or "tail," is `xs`.

All of these, though, are conventions, not definite rules. While we will generally adhere to the conventions in this book, any Haskell code you see out in the wild may not. Calling a type variable `x` instead of `a` is not going to break anything. As in the lambda calculus, the

names don't have any inherent meaning. We offer this information as a descriptive guide of Haskell conventions, not as rules you must follow in your own code.

Chapter 5

Types

She was the single artificer of the world
In which she sang. And when she sang, the sea,
Whatever self it had, became the self
That was her song, for she was the maker.

Wallace Stevens, "The Idea of Order at Key West"

5.1 Types

In the last chapter, we looked at some built-in datatypes, such as `Bool` and tuples, and had a brief run-in with the type classes `Num` and `Eq`. However, a deep understanding of types and how to read and interpret them is fundamental to reading and writing Haskell.

As we have seen, a datatype declaration defines a type constructor and one or more data constructors. Data constructors are the values of a particular type; they are also functions that let us create data, or values, of a particular type, although it will take some time before the full import of this becomes clear. In Haskell, you cannot create untyped data, so except for a sprinkling of syntactic sugar for things like numbers and functions, everything originates in a data constructor from some definition of a type.

In this chapter, we're going to take a deeper look at the type system and:

- Learn more about querying and reading type signatures.
- See that currying, unfortunately, has nothing to do with food.
- Take a closer look at different kinds of polymorphism.
- Look at type inference and how to declare types for our functions.

5.2 What are types for?

Haskell is an implementation of a pure lambda calculus, in the sense that it isn't much more than syntactic sugar over the basic system of variables, abstractions, and applications that constitute the rules of the lambda calculus—at least, of a *typed* lambda calculus. Developments in logic, mathematics, and computer science led to the discovery (or invention, take your pick) of a typed lambda calculus called *System F* in the 1970s. Haskell has improved on System F in some key ways, such as by allowing general recursion (more on that in a later chapter) and by implementing the Hindley-Milner system for type inference (more on that later in this chapter), but the core logic is the same.

So, why do we want types? Type systems in logic and mathematics have been designed to impose constraints that enforce correctness.

For our purposes, we can say that well-designed type systems help eliminate some classes of errors as well as concerns such as what the effect of a conditional over a non-Boolean value might be. A type system defines the associations between different parts of a program and checks that all the parts fit together in a logically consistent, provably correct way.

Let's consider a short, somewhat oversimplified example. The `Bool` type is a set with two inhabitants, `True` and `False`, as we saw in the last chapter. Wherever the values `True` and `False` occur in a Haskell program, the type checker will know they're members of the `Bool` type. The inverse is that whenever `Bool` is declared in a type signature, the compiler will expect one of those two values and only one of those two values; you get a type error if you try to pass a number where a `Bool` is expected.

In Haskell, where typing is *static*, type checking occurs *at compile time*. This means that many errors will be caught before you try to execute, or run, your program. The difference isn't always obvious, because GHCi allows you to type check your code interactively, as you're writing it, as well as execute it if it type checks. No type system can eliminate all possibilities for error, however, so runtime errors and exceptions can still occur, and therefore it is still necessary to test your programs. But the type system reduces the number and kinds of tests you need to write.

Good type systems can also enable compiler optimizations, because the compiler can know and predict certain things about the execution of a program based on the types. Types can also serve as documentation of your program, which is one reason we encourage you to declare types (that is, write the type signatures) for your functions. It won't matter too much when you're writing small programs, but as your programs get longer, type signatures can help you read your code and remember what you were doing—and help anyone else who might be trying to interpret your code, as well.

You may feel that Haskell's type system requires a lot of upfront work. This upfront cost comes with a later payoff: code that is safer and, down the line, easier to maintain. Working with a good type system can eliminate those tests that only check that you're passing the right sort of data around, and since tests are more code that you have to write (correctly) and maintain, it will eventually save you

time and effort.

Many, perhaps most, programming languages have type systems that feel like haggling with a petty merchant. However, we believe Haskell provides a type system that more closely resembles a quiet, pleasant conversation with a colleague than an argument in the bazaar. Much of what we suggest with regards to putting code in a file, loading it in a REPL, querying types in the REPL, and so forth, is about creating habits conducive to having this pleasant back and forth chat with your type system.

5.3 How to read type signatures

In previous chapters, we've seen that we can query types in the REPL with the `:type` or `:t` command. You can query types for functions, partially applied functions, and values—which are, in a way, fully applied functions.

When we query the types of values, we see something like this:

```
Prelude> :type 't'
't' :: Char
-- 't' has the type Char
Prelude> :type "julie"
"julie" :: [Char]
-- "julie" has the type String
Prelude> :type True
True :: Bool
-- True has the type Bool
```

When we query the types of numeric values, we see type class information instead of a concrete type, because the compiler doesn't know which specific numeric type a value is until the type is either declared or the compiler is forced to infer a specific type based on the function. For example, 13 may look like an integer to us, but that would only allow us to use it in computations that take integers (and not, say, in fractional division). For that reason, the compiler gives it the type with the broadest applicability (most polymorphic) and says it's a *constrained* polymorphic value with the type `Num a => a`:

```
Prelude> :type 13
13 :: Num a => a
```

We can give it a concrete type by declaring it:

```
Prelude> x = 13 :: Integer
Prelude> :t x
x :: Integer
```

You can also query the type signatures of functions, as we've seen:

```
Prelude> :type not
not :: Bool -> Bool
```

This takes one input of a `Bool` value and returns one `Bool` value. Given that type, there aren't too many things it even *could* do.[1]

Understanding the function type

The arrow, `->`, is the type constructor for functions in Haskell. It's baked into the language, but syntactically it works in very much the same way as all the other types you've seen so far. It's a type constructor, like `Bool`, except the `->` type constructor takes arguments and has no data constructors:

```
Prelude> :info (->)
data (->) a b
-- some further information is elided
```

If you compare this to the type constructor for the 2-tuple, you see the similarity:

```
Prelude> :info (,)
data (,) a b = (,) a b
```

We saw earlier that the tuple constructor needs to be applied to two values in order to construct a tuple. A function must similarly have two arguments—one input and one result—in order to be a function. Unlike the tuple constructor, though, the function type has no data constructors. The value that shows up at term level is the function. *Functions are values*.

[1] Four, to be precise. But if we assume that the standard `Prelude` functions are generally useful, it helps narrow it down considerably.

As we've said, the hallmark of a function is that it can be *applied*, and the structure of the type demonstrates this. The arrow is an *infix operator* that has two parameters and associates to the right (although function *application* is left associative). The parameterization suggests that we will apply the function to some argument that will be bound to the first parameter, with the second parameter, `b`, representing the return or result type. We will cover these things in more detail throughout this chapter.

Let's return to reading type signatures. The function `fst` is a value of type `(a, b) -> a`, where `->` is an infix type constructor that takes two arguments:

```
fst :: (a, b)  ->    a
--      [1]    [2]  [3]
```

1. The first parameter of `fst` has the type `(a, b)`. Note that the tuple type itself `(,)` takes two arguments, `a` and `b`.
2. The function type, `(->)`, has two parameters. One is `(a, b)`, and one is the result `a`.
3. The result of the function, which has type `a`. It's the same `a` that was in the tuple `(a, b)`.

How do we know it's the same `a`? We can say we know that the input type `a` and the output type `a` must be the same *type*, and we can see that nothing *happens* between the input and the output; that is, there is no operation that comes between them that could transform that `a` into some other value of that type.

Let's look at another function:

```
Prelude> :type length
length :: [a] -> Int
```

The `length` function takes one argument that is a list—note the square brackets—and returns an `Int` result. The `Int` result, in this case, will be the number of items in the list. The type of the inhabitants of the list is left unspecified; this function does not care—in fact, cannot care—what type of values are inside the list.

Type class-constrained type variables

Next, let's look at the types of some arithmetic functions. You may recall that the act of wrapping an infix operator in parentheses allows us to use the function just like a normal prefix function, including being able to query the type:

```
Prelude> :type (+)
(+) :: Num a => a -> a -> a
Prelude> :type (/)
(/) :: Fractional a => a -> a -> a
```

To describe these casually, we could say addition takes one numeric argument, adds it to a second numeric argument of the same type, and returns a numeric value of the same type as a result. Similarly, the fractional division function takes a fractional value, divides it by a second fractional value, and returns a third fractional value as a result. This isn't precise, but it will do for now.

The compiler gives the least specific and most general type it can. Instead of limiting this function to a concrete type, we get a type class-constrained polymorphic type variable. We'll save a fuller explanation of type classes for the next chapter. What we need to know here is that each type class offers a standard set of functions that can be used across several concrete types. When a type class is constraining a type variable in this way, the variable could represent any of the concrete types that have instances of that type class, so that specific operations on which the function depends are defined for that type. We say it's *constrained*, because we still don't know the concrete type of `a`, but we do know it can *only* be one of the types that has the required type class instance.

This generalization of numberhood is what lets us use the same numeric literals to represent numbers of different types. We can start with a polymorphic `Num a => a` value and then create specific kinds of numbers with a concrete type using the `::` to assign a type to the value:

```
Prelude> fifteen = 15
Prelude> :t fifteen
fifteen :: Num a => a
Prelude> fifteenInt = fifteen :: Int
```

```
Prelude> fifteenDouble = fifteen :: Double
Prelude> :t fifteenInt
fifteenInt :: Int
Prelude> :t fifteenDouble
fifteenDouble :: Double
```

We went from `Num a => a` to `Int` and `Double`. This works, because `Int` and `Double` each have an instance of the `Num` type class:

```
Prelude> :info Num
-- irrelevant bits elided
instance Num Int
instance Num Double
```

Since they both have instances of `Num`, the operations from `Num`, such as addition, are defined for both of them:

```
Prelude> fifteenInt + fifteenInt
30
Prelude> fifteenDouble + fifteenDouble
30.0
```

We can also make more specific versions of the `Num a => a` value we named `fifteen`, by using it in a way that requires it to become something more specific:

```
Prelude> fifteenDouble + fifteen
30.0
Prelude> fifteenInt + fifteen
30
```

What we cannot do is this:

```
Prelude> fifteenDouble + fifteenInt

• Couldn't match expected type ‘Double’
    with actual type ‘Int’
• In the second argument of ‘(+)’,
    namely ‘fifteenInt’
  In the expression:
    fifteenDouble + fifteenInt
```

```
In an equation for ‘it’:
  it = fifteenDouble + fifteenInt
```

We can't add those two values, because their types are no longer polymorphic, and their concrete types are different, so they have different definitions of how to implement addition. The type error message contrasts the *actual type* with the *expected type*. The actual type is what we provide; the expected type is what the compiler expects. Since we have `fifteenDouble` as our first argument, it expects the second value to also have the type `Double`, but it *actually* has the type `Int`.

A type signature might have multiple type class constraints on one or more of the variables. You will sometimes see (or write) type signatures such as:

```
(Num a, Num b) => a -> b -> b

-- or

(Ord a, Num a) => a -> a -> Ordering
```

Here, the constraints look like a tuple, although they don't add another function argument that you must provide, and they don't appear as a tuple at the value or term level. Nothing to the left of the type class arrow, `=>`, shows up at term level. The tuple of constraints *does* represent a product, or conjunction, of constraints, however.

In the first example above, there are two constraints, one for each variable. Both `a` and `b` must have instances of the `Num` type class. In the second example, both of the constraints are on the one variable `a`—that is, `a` must be a type that implements *both* `Ord` and `Num`.

Exercises: Type matching

Below, you'll find a list of several standard functions we've talked about previously. Under that is a list of their type signatures. Match the function to its type signature. Try to do it without peeking at the type signatures (either in the text or in GHCi), and then check your work. You may find it easier to start from the types and work out what you think a function of that type would do.

1. Functions:

   a) `not`

   b) `length`

   c) `concat`

   d) `head`

   e) `(<)`

2. Type signatures:

   a) `_ :: [a] -> a`

   b) `_ :: [[a]] -> [a]`

   c) `_ :: Bool -> Bool`

   d) `_ :: [a] -> Int`

   e) `_ :: Ord a => a -> a -> Bool`

5.4 Currying

As in the lambda calculus, arguments (*plural*) are a shorthand for the truth in Haskell: all functions in Haskell take one argument and return one result. Other programming languages, if you have any experience with them, typically allow you to define functions that can take multiple arguments. There is no support for this built into Haskell. Instead, there are syntactic conveniences that construct *curried* functions by default. Currying refers to the nesting of multiple functions, each accepting one argument and returning one result, which creates the illusion of multiple-parameter functions.

The arrows we've seen in type signatures denote the function type. We looked at the datatype definition earlier, but let's review:

```
data (->) a b
```

In order to have a function, you must have one input, the `a`, to apply the function to, and you'll get one result, the `b`, back. Each arrow in a type signature represents one argument and one result, with the final type being the final result. If you are constructing a function that requires multiple parameters, then the `b` can be another

function (the `a` can be another function as well). In that case, just like in lambda abstractions that have multiple heads, they are nested.

Let's break this down by looking at the type signature for addition, a function that requires multiple inputs:

```
(+) :: Num a => a -> a -> a
--     [  1  ]

(+) :: Num a => a -> a -> a
--              [   2   ]

(+) :: Num a => a -> a -> a
--                      [3]
```

1. This is the type class constraint that says `a` must have an instance of `Num`. Addition is defined in the `Num` type class.
2. The boundaries indicated above demarcate what you might call the two parameters to the function `+`, but all functions in Haskell take one argument and return one result. This is because functions in Haskell are nested like Matryoshka dolls in order to accept "multiple" arguments. The way the `->` type constructor for functions works means `a -> a -> a` represents successive function applications, each taking one argument and returning one result. The difference is that the function at the outermost layer is returning *another* function that accepts the next argument. This is called currying.
3. This is the result type for this function. It will be a number of the same type as the two inputs.

The way the type constructor for functions, `->`, is defined makes currying the default in Haskell. This is because it is an infix operator and right associative. Since it associates to the right, types are implicitly parenthesized like so:

```
f :: a -> a -> a

-- associates to

f :: a -> (a -> a)
```

and

```
map :: (a -> b) -> [a] -> [b]

-- associates to

map :: (a -> b) -> ([a] -> [b])
```

Let's see if we can unpack the notion of a right-associating infix operator giving us curried functions. The association here, or grouping into parentheses, is not to control precedence or order of evaluation; it only serves to group the parameters into arguments and results, since there can only be one argument and one result per arrow. Since all the arrows have the same precedence, the associativity does not change the precedence or order of evaluation.

Remember, when we have a lambda expression that appears to have two parameters, they are actually nested lambdas. Applying the expression to one argument returns a function that awaits application to a second argument. After you apply it to a second argument, you have a final result. You can nest more lambdas than two, of course, but the process is the same: one argument, one result, even though that result may be a function awaiting application to another argument.

The type constructor for functions and the types we see above are the same thing, but written in Haskell. When there are "two arguments" in Haskell, we apply our function to an argument, just as when we apply a lambda expression to an argument, and then return a result that is also a function and needs to be applied to a second argument.

Explicit parenthesization, as when an input parameter is itself a function (such as in `map`, above), may be used to indicate order of evaluation, but the implicit associativity of the function type does not mean the inner or final set of parentheses, i.e., the result type, evaluates *first*. Application is evaluation; in other words, the only way to evaluate anything is by applying functions, and function application is *left* associative. So, the leftmost, or outermost, arguments will be evaluated first, assuming anything gets evaluated (since Haskell is non-strict, you can't assume that anything will be evaluated at all, but this will be clarified later).

Partial application

Currying may be interesting, but many people wonder what the practical effect or value of currying is. We'll look now at a strategy called *partial application* to see what currying does for us. It's something we'll explore more as we go through the book, as well.

We use the double colon to assign a type. Making the type concrete will eliminate the type class constraint:

```
addStuff :: Integer -> Integer -> Integer
addStuff a b = a + b + 5
```

So, `addStuff` appears to take two `Integer` arguments and return an `Integer` result. But after loading that in GHCi, we see that it takes one argument and returns a function that takes one argument and returns one result:

```
Prelude> :t addStuff
addStuff :: Integer -> Integer -> Integer
Prelude> addTen = addStuff 5
Prelude> :t addTen
addTen :: Integer -> Integer
Prelude> fifteen = addTen 5
Prelude> fifteen
15
Prelude> addTen 15
25
Prelude> addStuff 5 5
15
```

Here, `fifteen` is equal to `addStuff 5 5`, because `addTen` is equal to `addStuff 5`. The ability to apply only some of a function's arguments is called partial application. This lets us reuse `addStuff` and create a new function from it with one of the arguments applied.

If we recall that `->` is a type constructor and associates to the right, this becomes clearer:

```
addStuff :: Integer -> Integer -> Integer

-- with explicit parenthesization

addStuff :: Integer -> (Integer -> Integer)
```

Applying `addStuff` to one `Integer` argument gives us the function `addTen`, which is the return function of `addStuff`. Applying `addTen` to an `Integer` argument gives us a return value, so the type of `fifteen` is `Integer`—no more function arrows.

Let's check our understanding of partial application with a function that isn't commutative:

```
subtractStuff :: Integer
              -> Integer
              -> Integer
subtractStuff x y = x - y - 10
subtractOne = subtractStuff 1
```

```
Prelude> :t subtractOne
subtractOne :: Integer -> Integer
Prelude> result = subtractOne 11
Prelude> result
-20
```

Why did we get this result? Because of the order in which we applied arguments, `result` is equal to `1 - 11 - 10`.

Manual currying and uncurrying

Functions in Haskell are curried by default, but it is possible to "uncurry" them. *Uncurrying* means, for example, un-nesting two functions and replacing them with a tuple of two values (these would be the two values you want to use as arguments). If you uncurry the `+` function, the type changes from `Num a => a -> a -> a` to the type `Num a => (a, a) -> a`, which better fits the description "takes two arguments and returns one result" than a curried function. Some older functional languages default to using product types like tuples to express multiple arguments. To sum up:

- Uncurried functions: One function, many arguments.
- Curried functions: Many functions, one argument apiece.

You can also desugar the automatic currying yourself, by nesting the arguments with lambdas, though there's rarely a reason to do so.

We'll use anonymous lambda syntax here to show you some examples of uncurrying. You may want to review anonymous lambda syntax or try comparing these functions directly and thinking of the backslash as a lambda:

```
nonsense :: Bool -> Integer
nonsense True = 805
nonsense False = 31337

curriedFunction :: Integer
                -> Bool
                -> Integer
curriedFunction i b =
  i + (nonsense b)

uncurriedFunction :: (Integer, Bool)
                  -> Integer
uncurriedFunction (i, b) =
  i + (nonsense b)

anonymous :: Integer -> Bool -> Integer
anonymous = \i b -> i + (nonsense b)

anonNested :: Integer
           -> Bool
           -> Integer
anonNested =
  \i -> \b -> i + (nonsense b)
```

Then, when we test the functions in the REPL:

```
Prelude> curriedFunction 10 False
31347
Prelude> anonymous 10 False
```

```
31347
Prelude> anonNested 10 False
31347
```

They are all the same function, all giving the same result. In `anonNested`, we manually nest the anonymous lambdas to get a function that is semantically identical to `curriedFunction` but doesn't leverage the automatic currying.

Currying and uncurrying existing functions

It turns out, we can curry and uncurry functions with multiple parameters generically without writing new code for each one. Consider the following example of manual currying:

```
Prelude> curry f a b = f (a, b)
Prelude> :t curry
curry :: ((t1, t2) -> t) -> t1 -> t2 -> t
Prelude> :t fst
fst :: (a, b) -> a
Prelude> :t curry fst
curry fst :: t -> b -> t
Prelude> fst (1, 2)
1
Prelude> curry fst 1 2
1
```

Then, of uncurrying:

```
Prelude> uncurry f (a, b) = f a b
Prelude> :t uncurry
uncurry :: (t1 -> t2 -> t) -> (t1, t2) -> t
Prelude> :t (+)
(+) :: Num a => a -> a -> a
Prelude> (+) 1 2
3
Prelude> uncurry (+) (1, 2)
3
```

Currying and uncurrying functions of three or more arguments automatically is also possible but trickier. We'll leave that be, but investigate on your own if you like.

Sectioning

We mentioned sectioning in Chapter 2, and now that we've talked a bit more about currying and partial application, it may be more clear what's happening there. The term *sectioning* specifically refers to the partial application of infix operators, which has a special syntax that allows you to choose whether you're partially applying the operator to the first or the second argument:

```
Prelude> x = 5
Prelude> y = (2^)
Prelude> z = (^2)
Prelude> y x
32
Prelude> z x
25
```

With commutative functions such as addition, the argument order does not matter. We will usually section addition as, for example, `(+3)`, but when we start using partially applied functions a lot with maps and folds and so forth, you'll be able to see the difference that the argument order can make with non-commutative operators.

This does not only work with arithmetic, though:

```
Prelude> celebrate = (++ " woot!")
Prelude> celebrate "naptime"
"naptime woot!"
Prelude> celebrate "dogs"
"dogs woot!"
```

You can also use this syntax with functions that are normally prefix if you use backticks to make them infix (note that the `..` in the code below is a shorthand for constructing a list of all the elements from the first to the last value given—go ahead and play with this in your REPL):

```
Prelude> elem 9 [1..10]
True
Prelude> 9 `elem` [1..10]
True
Prelude> c = (`elem` [1..10])
Prelude> c 9
True
Prelude> c 25
False
```

If you partially apply `elem` in its usual prefix form, then the argument you apply it to would necessarily be the first argument:

```
Prelude> hasTen = elem 10
Prelude> hasTen [1..9]
False
Prelude> hasTen [5..15]
True
```

Partial application is common enough in Haskell that, over time, you'll develop an intuition for it. The sectioning syntax exists to allow you some freedom as to which argument of a binary operator you apply the function to.

Exercises: Type arguments

Given a function and its type, tell us what type results from applying some or all of the arguments.

You can check your work in the REPL like this (using the first question as an example):

```
Prelude> u = undefined
Prelude> f :: a -> a -> a -> a; f = u
Prelude> x :: Char; x = u
Prelude> :t f x
```

It turns out that you can check the types of things that aren't implemented yet, so long as you give GHCi an `undefined` to bind the signature to.

1. If the type of `f` is `a -> a -> a -> a`, and the type of `x` is `Char`, then the type of `f x` is:

   a) `Char -> Char -> Char`

   b) `x -> x -> x -> x`

   c) `a -> a -> a`

   d) `a -> a -> a -> Char`

2. If the type of `g` is `a -> b -> c -> b`, then the type of `g 0 'c' "woot"` is:

   a) `String`

   b) `Char -> String`

   c) `Int`

   d) `Char`

3. If the type of `h` is `(Num a, Num b) => a -> b -> b`, then the type of `h 1.0 2` is:

   a) `Double`

   b) `Integer`

   c) `Integral b => b`

   d) `Num b => b`

   Note that because the type variables `a` and `b` are different, the compiler *must* assume that the types could be different.

4. If the type of `h` is `(Num a, Num b) => a -> b -> b`, then the type of `h 1 (5.5 :: Double)` is:

   a) `Integer`

   b) `Fractional b => b`

   c) `Double`

   d) `Num b => b`

5. If the type of `jackal` is `(Ord a, Eq b) => a -> b -> a`, then the type of `jackal "keyboard" "has the word jackal in it"` is:

   a) `[Char]`

b) `Eq b => b`

c) `b -> [Char]`

d) `b`

e) `Eq b => b -> [Char]`

6. If the type of `jackal` is `(Ord a, Eq b) => a -> b -> a`, then the type of `jackal "keyboard"` is:

a) `b`

b) `Eq b => b`

c) `[Char]`

d) `b -> [Char]`

e) `Eq b => b -> [Char]`

7. If the type of `kessel` is `(Ord a, Num b) => a -> b -> a`, then the type of `kessel 1 2` is:

a) `Integer`

b) `Int`

c) `a`

d) `(Num a, Ord a) => a`

e) `Ord a => a`

f) `Num a => a`

8. If the type of `kessel` is `(Ord a, Num b) => a -> b -> a`, then the type of `kessel 1 (2 :: Integer)` is:

a) `(Num a, Ord a) => a`

b) `Int`

c) `a`

d) `Num a => a`

e) `Ord a => a`

f) `Integer`

9. If the type of `kessel` is `(Ord a, Num b) => a -> b -> a`, then the type of `kessel (1 :: Integer) 2` is:

a) `Num a => a`

b) `Ord a => a`

c) `Integer`

d) `(Num a, Ord a) => a`

e) `a`

5.5 Polymorphism

Polymorph is a word of relatively recent provenance. It was invented in the early 19th century from the Greek words *poly* for "many" and *morph* for "form." The *-ic* suffix in polymorphic means "made of." So, "polymorphic" means "made of many forms." In programming, this is understood to be in contrast to *monomorphic*, "made of one form."

Polymorphic type variables give us the ability to implement expressions that can accept arguments and return results of different types without having to write variations on the same expression for each type. It would be inefficient if you were doing arithmetic and had to write the same code over and over for different numeric types. The good news is that the numerical functions that come with your GHC installation and the `Prelude` are polymorphic by default. Broadly speaking, type signatures may have three kinds of types: concrete, constrained polymorphic, or parametrically polymorphic.

In Haskell, polymorphism divides into two categories: *parametric polymorphism* and *constrained polymorphism*. If you've encountered polymorphism before, it was probably a form of constrained, often called ad-hoc, polymorphism. Ad-hoc polymorphism[2] in Haskell is implemented with type classes.

Parametric polymorphism is broader than ad-hoc polymorphism. Parametric polymorphism refers to type variables, or *parameters*, that are fully polymorphic. When unconstrained by a type class, their final, concrete type could be anything. Constrained polymorphism, on the other hand, puts type class constraints on a variable, decreasing the number of concrete types it could be, but increasing what you can do with it by defining and bringing into scope a set of specific operations.

[2] See Philip Wadler and Stephen Blott's paper, *How to make ad-hoc polymorphism less ad hoc*: `http://people.csail.mit.edu/dnj/teaching/6898/papers/wadler88.pdf`.

Recall that when you see a lowercase name in a type signature, it is a type variable and polymorphic (like `a`, `t`, etc.). If the type is capitalized, it is a specific, concrete type such as `Int`, `Bool`, etc.

Let's consider a parametrically polymorphic function: *identity*. The `id` function comes with the `Prelude` and is called the identity function, because it is the identity for any value in our language. In the next example, the type variable `a` is parametrically polymorphic and not constrained by a type class. Passing any value to `id` will return the same value:

```
id :: a -> a
```

This type says: for all `a`, get an argument of some type `a` and return a value of the same type `a`.

This is the maximally polymorphic signature for `id`. It allows this function to work with any type of data:

```
Prelude> id 1
1
Prelude> id "blah"
"blah"
Prelude> inc = (+1)
Prelude> inc 2
3
Prelude> (id inc) 2
3
```

Based on the type of `id`, we are guaranteed that this will be its behavior—it cannot do anything else! The `a` in the type signature cannot change, because the type variable gets fixed to a concrete type throughout the entire type signature (`a == a`). If one applies `id` to a value of type `Int`, the `a` is fixed to type `Int`. By default, type variables are resolved at the left-most part of the type signature and are fixed once sufficient information to bind them to a concrete type is available.

The arguments in parametrically polymorphic functions, like `id`, could be anything, any type or type class, so the terms of the function are more restricted, because there are no methods[3] or information

[3] A *method* is a function associated with a type class. You'll see more on this later.

attached to them. A function with the type `id :: a -> a` cannot do anything other than return `a`, because there is no information or method attached to its parameter at all—in other words, since we don't have any functions that can do anything interesting with a totally generic value, nothing can be done *with* `a`. On the other hand, a function like `negate`, with the similar-appearing type signature of `Num a => a -> a`, constrains the `a` variable to be an instance of the `Num` type class. In this case, `a` has fewer concrete types it could be, but there is a set of methods you can use, a set of things that can be done with `a`.

If a variable represents a set of possible values, then a type variable represents a set of possible types. When there is no type class constraint, the set of possible types a variable could represent is effectively unlimited. Type class constraints limit the set of potential types (and, thus, potential values) while also passing along the common functions that can be used with those values.

Concrete types have even more flexibility in terms of computation. This has to do with the additive nature of type classes. For example, an `Int` is only an `Int`, but it can make use of the methods of the `Num` *and* `Integral` type classes, because it has instances of both. We can describe `Num` as a *superclass* of several other numeric type classes that all inherit operations from `Num`.

In sum, if a variable could be *anything*, then there's little that can be done to it, because it has no specific methods. If it can be *some* types (say, a type that has an instance of `Num`), then it has some methods. If it is a concrete type, you lose the type flexibility but, due to the additive nature of type class inheritance, gain more potential methods. It's important to note that this inheritance extends downward from a superclass, such as `Num`, to subclasses, such as `Integral` and then `Int`, but not the other way around. That is, if something has an instance of `Num` but not an instance of `Integral`, it can't implement the methods of the `Integral` type class. A subclass cannot override the methods of its superclass.

A function is polymorphic when its type signature has variables that can represent more than one type. That is, its parameters are polymorphic. Parametric polymorphism refers to fully polymorphic (unconstrained by a type class) parameters. Parametricity is the property we get from having parametric polymorphism. *Para-*

metricity means that the behavior of a function with respect to the types of its (parametrically polymorphic) arguments is uniform. The behavior *cannot* change just because it was applied to an argument of a different type.

Exercises: Parametricity

All you can do with a parametrically polymorphic value is pass or not pass it to some other expression. Prove that to yourself with these small demonstrations.

1. Given the type `a -> a`, which is the type for `id`, attempt to make a function that terminates successfully but that does something other than return the same value. This is impossible, but you should try it anyway.
2. We can get a more comfortable appreciation of parametricity by looking at `a -> a -> a`. This hypothetical function `a -> a -> a` has only two implementations. Write both possible versions of `a -> a -> a`. After doing so, try to violate the constraints of parametrically polymorphic values we outlined above.
3. Implement `a -> b -> b`. How many implementations can it have? Does its behavior change when the types of `a` and `b` change?

Polymorphic constants

We've seen that there are several types of numbers in Haskell and that there are restrictions on using different types of numbers in different functions. But intuitively, we also see it would be odd if we could not do arithmetic along the lines of `-10 + 6.3`. Well, let's try it:

```
Prelude> (-10) + 6.3
-3.7
```

That works just fine. Why? Let's look at the types and see if we can find out:

```
Prelude> :t (-10) + 6.3
(-10) + 6.3 :: Fractional a => a
Prelude> :t (-10)
(-10) :: Num a => a
```

Numeric literals like `-10` and `6.3` are polymorphic and remain so until given a more specific type. `Num a` and `Fractional a` are different type class constraints, and the `a` is the type variable in scope. In the type for the entire equation, we see that the compiler infers that it is working with `Fractional` numbers. It has to, in order to accommodate the fractional number `6.3`. Fine, but what about `-10`? We see that the type of `-10` is given maximum polymorphism by only being an instance of the `Num` type class, which could be any type of number whatsoever. We call this a polymorphic constant; `-10` is not a variable, of course, but the type that it instantiates could be any numeric type, so its underlying representation is polymorphic. It will have to resolve into a concrete type at some point in order to evaluate, however.

We can force the compiler to be more specific about the types of numbers by declaring them ourselves:

```
Prelude> x = 5 + 5
Prelude> :t x
x :: Num a => a
Prelude> x = 5 + 5 :: Int
Prelude> :t x
x :: Int
```

In the first example, we do not specify a type for the numbers, so the type signature defaults to the broadest interpretation, but in the second version, we tell the compiler to use the `Int` type.

Working around constraints

Previously, we looked at a function called `length` that takes a list and counts the number of members and returns that number as an `Int` value. We saw in the last chapter that because `Int` is not a `Fractional` number, this function won't work:

```
Prelude> 6 / length [1, 2, 3]

• No instance for (Fractional Int) arising
    from a use of ‘/’
• In the expression: 6 / length [1, 2, 3]
  In an equation for ‘it’:
```

```
it = 6 / length [1, 2, 3]
```

Here, the problem is that `length` isn't polymorphic enough. The type class `Fractional` includes several types of numbers, but `Int` isn't one of them, and that's all `length` can return. Haskell does offer ways to work around this type of conflict, though. In this case, we will use a function called `fromIntegral` that takes an integral value and forces it to implement the `Num` type class, rendering it polymorphic. Here's what the type signature looks like:

```
Prelude> :type fromIntegral
fromIntegral ::
  (Num b, Integral a) => a -> b
```

So, it takes a value, `a`, of an `Integral` type and returns it as a value, `b`, of any `Num` type. Let's see how that works with our fractional division problem:

```
Prelude> xs = [1, 2, 3]
Prelude> 6 / fromIntegral (length xs)
2.0
```

And now all is right with the world once again.

5.6 Type inference

Haskell does not obligate us to assert a type for every expression or value in our programs, because it has *type inference*. Type inference is an algorithm for determining the types of expressions. Haskell's type inference is built on an extended version of the Damas-Hindley-Milner type system.

Haskell will infer the most generally applicable (polymorphic) type that is still correct. Essentially, the compiler starts from the values that have types it knows and then works out the types of the other values. As you mature as a Haskell programmer, you'll find this is principally useful for when you're still figuring out new code rather than for code that is "done." Once your program is "done," you will certainly know the types of all the functions, and it's considered good practice to explicitly declare them. Remember when we suggested that a good type system is like a pleasant conversation with a colleague? Think

of type inference as a helpful colleague working through a problem with you.

For example, we can write `id` ourselves:

```
Prelude> ourId x = x
Prelude> :t ourId
ourId :: t -> t
Prelude> ourId 1
1
Prelude> ourId "blah"
"blah"
```

Here, we let GHCi infer the type of `ourId` itself. Due to alpha equivalence, the difference in letters (`t` here versus `a` above) makes no difference at all. Type variables have no meaning outside of the type signatures where they are bound.

For this function, we again ask the compiler to infer the type:

```
Prelude> myGreet x = x ++ " Julie"
Prelude> myGreet "hello"
"hello Julie"
Prelude> :type myGreet
myGreet :: [Char] -> [Char]
```

The compiler knows the function `++` and has one value to work with already that it knows is a `String`. It doesn't have to work very hard to infer a type signature from that information. If, however, we take out the string value and replace it with another variable, look what happens:

```
Prelude> myGreet x y = x ++ y
Prelude> :type myGreet
myGreet :: [a] -> [a] -> [a]
```

We're back to a polymorphic type signature, the same signature for `++` itself, because the compiler has no information by which to infer the types for any of those variables (other than that they are lists of some sort).

Let's see type inference at work. Open your editor of choice and enter the following snippet:

```
-- typeInference1.hs
module TypeInference1 where

f :: Num a => a -> a -> a
f x y = x + y + 3
```

Then, load the code into GHCi to experiment:

```
Prelude> :l typeInference1.hs
[1 of 1] Compiling TypeInference1
Ok, one module loaded.
Prelude> f 1 2
6
Prelude> :t f
f :: Num a => a -> a -> a
Prelude> :t f 1
f 1 :: Num a => a -> a
```

Because the numeric literals in Haskell have the (type class constrained) polymorphic type `Num a => a`, we don't get a more specific type when applying `f` to 1.

Look at what happens when we elide the explicit type signature for `f`:

```
-- typeInference2.hs
module TypeInference2 where

f x y = x + y + 3
```

No type signature for `f`, so does it compile? Does it work?

```
Prelude> :l typeInference2.hs
[1 of 1] Compiling TypeInference2
Ok, one module loaded.
Prelude> :t f
f :: Num a => a -> a -> a
Prelude> f 1 2
6
```

Nothing changes. In certain cases, there might be a change, usually when you are using type classes in a way that doesn't make it clear which type you mean unless you assert one.

Exercises: Apply yourself

Look at these pairs of functions. One function is unapplied, so the compiler will infer a maximally polymorphic type. The second function has been applied to a value, so the inferred type signature may have become concrete, or at least less polymorphic. Figure out how the type would change and why, make a note of what you think the new inferred type should be, and then check your work in GHCi.

1. Type signature of general function:

   ```
   (++) :: [a] -> [a] -> [a]
   ```

   How might that change when we apply it to the following value?

   ```
   myConcat x = x ++ " yo"
   ```

2. General function:

   ```
   (*) :: Num a => a -> a -> a
   ```

   Applied to a value:

   ```
   myMult x = (x / 3) * 5
   ```

3. ```
   take :: Int -> [a] -> [a]

   myTake x = take x "hey you"
   ```

4. ```
   (>) :: Ord a => a -> a -> Bool

   myCom x = x > (length [1..10])
   ```

5. ```
   (<) :: Ord a => a -> a -> Bool

   myAlph x = x < 'z'
   ```
   ```

5.7 Asserting types for declarations

Most of the time, we want to declare our types, rather than relying on type inference. Adding type signatures to your code can provide guidance to you as you write your functions. It can also help the compiler give you information about where your code is going wrong. As programs become longer and more complex, type signatures become even more important, as they help you or other programmers trying to use your code read it and figure out what it's supposed to do. This section will look at how to declare types. We will start with some trivial examples.

You may remember the `triple` function we saw before. If we allow the compiler to infer the type, we end up with this:

```
Prelude> triple x = x * 3
Prelude> :type triple
triple :: Num a => a -> a
```

Here, the `triple` function is made from the `*` function, which has the type `(*) :: Num a => a -> a -> a`, but we already apply one of the arguments, which is the `3`, so there is one fewer parameter in this type signature. It is still polymorphic, because it can't tell what type `3` is, yet. If, however, we want to ensure that our inputs and result can only be integers, this is how we declare that:

```
Prelude> triple x = x * 3 :: Integer
Prelude> :t triple
triple :: Integer -> Integer
```

Note the type class constraint is gone, because `Integer` implements `Num`, so that constraint is redundant.

Here's another example of a type declaration for our `triple` function; this one is more like what you would see in a source file:

```
-- type declaration
triple :: Integer -> Integer

-- function declaration
triple x = x * 3
```

This is how most Haskell code you look at will be laid out, with separate top-level declarations for types and functions. Such top-level declarations are in scope throughout a module.

It is possible, though uncommon, to declare types locally with `let` and `where`. Here's an example of assigning a type within a `where` clause:

```
triple x = tripleItYo x
  where tripleItYo :: Integer -> Integer
        tripleItYo y = y * 3
```

We don't have to assert the type of `triple`:

```
Prelude> :t triple
triple :: Integer -> Integer
```

The assertion in the `where` clause narrows our type down from `Num a => a -> a` to `Integer -> Integer`. GHC will pick up and propagate type information for inference from applications of functions, sub-expressions, definitions—almost anywhere. The type inference is strong with this one.

There *are* constraints on our ability to declare types, however. For example, if we try to make the `+` function return a `String`, we get an error message:

```
Prelude> x = 5 + 5 :: String

• No instance for (Num String) arising from
    a use of '+'
• In the expression: 5 + 5 :: String
  In an equation for
    'x': x = 5 + 5 :: String
```

This function cannot accept arguments of type `String`. In this case, it's overdetermined, both because the `+` function is limited to types implementing the `Num` type class and also because we've already passed it two numeric literals as values. The numeric literals could be any of several numeric types under the hood, but they can't be `String`, because `String` does not implement the `Num` type class.

5.8 Chapter exercises

Welcome to another round of "Knowing is not enough; we must apply."

Multiple choice

1. A value of type `[a]` is:
   a) a list of alphabetic characters
   b) a list of lists
   c) a list of elements that are all of some type `a`
   d) a list of elements that are all of different types
2. A function of type `[[a]] -> [a]` could:
   a) take a list of strings as an argument
   b) transform a character into a string
   c) transform a string into a list of strings
   d) take two arguments
3. A function of type `[a] -> Int -> a`:
   a) takes one argument
   b) returns one element of type `a` from a list
   c) must return an `Int` value
   d) is completely fictional
4. A function of type `(a, b) -> a`:
   a) takes a list argument and returns a `Char` value
   b) has zero arguments
   c) takes a tuple argument and returns the first value
   d) requires that `a` and `b` have different types

Determine the type

For the following functions, determine the type of the specified value. We suggest you type them into a file, and load the contents of the file in GHCi. In all likelihood, it initially will not have the polymorphic types you might expect due to the *monomorphism restriction*. That means that top-level declarations by default will have a concrete type if any can be determined. You can fix this by setting up your file like so:

```
{-# LANGUAGE NoMonomorphismRestriction #-}

module DetermineTheType where

-- simple example
example = 1
```

If you do not include the `NoMonomorphismRestriction` extension, `example` would have the type `Integer` instead of `Num a => a`. Do your best to determine the *most* polymorphic type an expression could have in the following exercises.

1. All function applications return a value. Determine the values returned by these function applications and the types of those values:

   a) `(* 9) 6`

   b) `head [(0,"doge"),(1,"kitteh")]`

   c) `head [(0 :: Integer ,"doge"),(1,"kitteh")]`

   d) `if False then True else False`

   e) `length [1, 2, 3, 4, 5]`

   f) `(length [1, 2, 3, 4]) > (length "TACOCAT")`

2. Given:

   ```
   x = 5
   y = x + 5
   w = y * 10
   ```

What is the type of `w`?

3. Given:

```
x = 5
y = x + 5
z y = y * 10
```

What is the type of `z`?

4. Given:

```
x = 5
y = x + 5
f = 4 / y
```

What is the type of `f`?

5. Given:

```
x = "Julie"
y = " <3 "
z = "Haskell"
f = x ++ y ++ z
```

What is the type of `f`?

Does it compile?

For each set of expressions, figure out which expression, if any, causes the compiler to squawk at you and why. Fix them if you can:

1. ```
   bigNum = (^) 5 $ 10
   wahoo = bigNum $ 10
   ```

2. ```
   x = print
   y = print "woohoo!"
   z = x "hello world"
   ```

3. ```
   a = (+)
   b = 5
   c = b 10
   d = c 200
   ```
   ```

4. ```
   a = 12 + b
   b = 10000 * c
   ```

Type variable or specific type constructor?

1. You will be shown a type declaration, and you should categorize each type. The choices are: a fully polymorphic type variable, a constrained polymorphic type variable, or a concrete type constructor.

   ```
   f :: Num a => a -> b -> Int -> Int
   --          [1]  [2]   [3]    [4]
   ```

   Here, the answers would be: constrained polymorphic, i.e. `Num` (0), fully polymorphic (1), and concrete (2 and 3).

2. Categorize each component of this type signature as described in the previous example:

   ```
   f :: zed -> Zed -> Blah
   ```

3. Categorize each component of this type signature:

   ```
   f :: Enum b => a -> b -> C
   ```

4. Categorize each component of this type signature:

   ```
   f :: f -> g -> C
   ```

Write a type signature

For the following expressions, please add a type signature. You should be able to rely on GHCi type inference to check your work, although you might not have precisely the same answer as GHCi gives (due to polymorphism, etc).

1. While we haven't fully explained this syntax yet, you saw it in Chapter 2 and as a solution to an exercise in Chapter 4. This syntax is a way of destructuring a single element of a list by pattern matching:

   ```
   functionH ::
   functionH (x:_) = x
   ```
   ```

2. ```
   functionC ::
   functionC x y =
     if (x > y) then True else False
   ```

3. ```
   functionS ::
   functionS (x, y) = y
   ```

Given a type, write the function

You will be shown a type and a function that needs to be written. Use the information the type provides to determine what the function should do. We'll also tell you how many ways there are to write the function. Syntactically different but semantically equivalent implementations are not counted as being different. For example, writing a function one way and then rewriting the semantically identical function but using anonymous lambda syntax does not count as two implementations.

To make things a little easier, we'll demonstrate how to solve this kind of exercise:

```
myFunc :: (x -> y)
       -> (y -> z)
       -> c
       -> (a, x)
       -> (a, z)
myFunc xToY yToZ _ (a, x) = undefined
```

Working through the above, we have a function that takes four arguments. The final result is a tuple with the type `(a, z)`. It turns out, the `c` argument is nowhere in our result, and there's nothing we need to do with it, so we use the underscore to ignore it. We name the two function arguments by their types and pattern match on the tuple argument. The only way to get the second value of the tuple from the type `x` to the type `z` is to use *both* of the functions furnished to us. If we try the following:

```
myFunc xToY yToZ _ (a, x) =
  (a, (xToY x))
```

We get a type error that the expected type is `z` but the actual type is `y`. That's because we're on the right path, but we're not quite done yet! Accordingly, the following should type check:

```
myFunc :: (x -> y)
       -> (y -> z)
       -> c
       -> (a, x)
       -> (a, z)
myFunc xToY yToZ _ (a, x) =
  (a, (yToZ (xToY x)))
```

1. There is only one function definition that type checks and doesn't go into an infinite loop when you run it:

   ```
   i :: a -> a
   i = undefined
   ```

2. There is only one version that works:

   ```
   c :: a -> b -> a
   c = undefined
   ```

3. Given alpha equivalence, are the variables `c''` and `c` (from the previous exercise) the same thing?

   ```
   c'' :: b -> a -> b
   c'' = ?
   ```

4. Only one version works:

   ```
   c' :: a -> b -> b
   c' = undefined
   ```

5. There are multiple possibilities, at least two of which you've seen in previous chapters:

   ```
   r :: [a] -> [a]
   r = undefined
   ```

6. Only one version will type check:

   ```
   co :: (b -> c) -> (a -> b) -> a -> c
   co = undefined
   ```

7. One version will type check:

```
a :: (a -> c) -> a -> a
a = undefined
```

8. One version will type check:

```
a' :: (a -> b) -> a -> b
a' = undefined
```

Fix it

Won't someone take pity on this poor, broken code and fix it up? Be sure to check carefully for things like capitalization, parentheses, and indentation:

1.
```
module sing where

fstString :: [Char] ++ [Char]
fstString x = x ++ " in the rain"

sndString :: [Char] -> Char
sndString x = x ++ " over the rainbow"

sing = if (x > y) then
         fstString x or sndString y
where x = "Singin"
      x = "Somewhere"
```

2. Now that it's fixed, make a minor change so that it sings the other song. If you're lucky, you'll end up with both songs stuck in your head!

3.
```haskell
-- arith3broken.hs
module Arith3Broken where

main :: IO ()
Main = do
  print 1 + 2
  putStrLn 10
  print (negate -1)
  print ((+) 0 blah)
  where blah = negate 1
```

Type-Kwon-Do

The name is courtesy of Phillip Wright.[4] Thank you for the idea!

The focus here is on manipulating terms in order to get the types to fit. This *sort* of exercise is something you'll encounter in writing real Haskell code, so practicing will make it easier to deal with when you get there. Training in the subtle art of type-kwon-do will make you better at writing ordinary code, as well.

We provide the types and bottomed out (declared as `undefined`) terms. *Bottom* and *undefined* will be explained in more detail later. The contents of the terms are irrelevant here. You'll use only the declarations provided and what the `Prelude` provides by default unless otherwise specified. Your goal is to make the `???`'d declaration pass the type checker by modifying it and it alone.

Here's an example of how we present these exercises and how you are expected to solve them:

```haskell
data Woot

data Blah

f :: Woot -> Blah
f = undefined

g :: (Blah, Woot) -> (Blah, Blah)
g = ???
```

[4] https://twitter.com/SixBitProxyWax

Above, it's the function `g` that you're supposed to implement; however, you can't evaluate anything. You're only allowed to use type checking and type inference to validate your answers. Also, note that we're using a trick for defining datatypes that can be named in a type signature but have no values. Here's an example of a valid solution:

```
g :: (Blah, Woot) -> (Blah, Blah)
g (b, w) = (b, f w)
```

The idea is to only fill in what we've marked with `???`. Note: not all terms will always be used in the intended solution to a problem!

1. ```
   f :: Int -> String
   f = undefined

   g :: String -> Char
   g = undefined

   h :: Int -> Char
   h = ???
   ```

2. ```
   data A
   data B
   data C

   q :: A -> B
   q = undefined

   w :: B -> C
   w = undefined

   e :: A -> C
   e = ???
   ```

3. ```
   data X
   data Y
   data Z

   xz :: X -> Z
   xz = undefined
   ```
   ```

```
yz :: Y -> Z
yz = undefined

xform :: (X, Y) -> (Z, Z)
xform = ???
```

4. ```
   munge :: (x -> y)
         -> (y -> (w, z))
         -> x
         -> w
   munge = ???
   ```

5.9 Definitions

1. *Polymorphism* refers to type variables that may refer to more than one concrete type. In Haskell, this will usually manifest as *parametric* or *ad-hoc* polymorphism. By having a larger set of types, we intersect their commonalities to produce a smaller set of correct terms. This makes it less likely that we'll write an incorrect program and lets us reuse the code with other types.

2. *Type inference* is a faculty some programming languages, most notably Haskell and ML, have to *infer* principal types from terms without requiring explicit type annotations. There are, in some cases, terms in Haskell that can be well-typed but which have no principal type. In those cases, an explicit type annotation must be added.

   With respect to Haskell, the *principal type* is the most generic type that still type checks. More generally, a *principal type* is a property of the type system you're interacting with. Principal typing holds for that type system if a type can be found for a term in an environment for which all other types for that term are instances of the principal type. Given the inferred types:

   ```
   a
   Num a => a
   Int
   ```

   The principal type here is the parametrically polymorphic `a`. Given these types:
   ```

```
(Ord a, Num a) => a
Integer
```

The principal type is `(Ord a, Num a) => a`.

3. A *type variable* is a way to refer to an unspecified type or set of types in Haskell type signatures. Type variables ordinarily will be equal to themselves throughout a type signature. Let us consider some examples:

```
id :: a -> a
```

The only type variable, `a`, occurs twice, once as an argument, once as a result. The type variable is parametrically polymorphic and could be strictly anything.

```
(+) :: Num a => a -> a -> a
```

Only one type variable named `a`, again. `a` is constrained to requiring an instance of `Num`. Two arguments of the same type `a` and one result.

4. A *type class* is a means of expressing faculties or interfaces that multiple datatypes may have in common. This enables us to write code exclusively in terms of those commonalities without repeating ourselves for each instance. Just as one may sum values of type `Int`, `Integer`, `Float`, `Double`, and `Rational`, we can avoid having different `+`, `*`, `-`, `negate`, etc. functions for each type by unifying them into a single type class. Importantly, these can then be used with *all* types that have a `Num` instance. Thus, a type class provides us a means to write code in terms of those operators and have our functions be compatible with all types that have instances of that type class, whether they already exist or are yet to be invented (by you, perhaps).

5. *Parametricity* is the property that holds in the presence of parametric polymorphism. Parametricity states that the behavior of a function will be uniform across all concrete applications of that function. Parametricity[5] tells us that the function:

[5]Examples are courtesy of the `@parametricity` twitter account: `https://twitter.com/parametricity`.

```
id :: a -> a
```

Can be understood to have the same, exact behavior for every type in Haskell without us needing to see how it was written. It is the same property that tells us that, according to this type signature:

```
const :: a -> b -> a
```

`const` *must* return the first value—parametricity and the definition of the type require it!

```
f :: a -> a -> a
```

Here, `f` can only return the first or second value, nothing else, and it will always return one or the other consistently without changing. If the function `f` made use of `+` or `*`, its type would necessarily be constrained by the type class `Num` and thus be an example of ad-hoc, rather than parametric, polymorphism.

```
blahFunc :: b -> String
```

`blahFunc` totally ignores its argument and is effectively a constant value of type `String` that requires a throw-away argument for no reason.

```
convList :: a -> [a]
```

Unless the result is `[]`, the resulting list has values that are all the same—and the list will always be the same length.

6. *Ad-hoc polymorphism* (sometimes called "constrained polymorphism") is polymorphism that applies one or more type class constraints to what would've otherwise been a parametrically polymorphic type variable. Here, rather than representing a uniformity of behavior across all concrete applications, the purpose of ad-hoc polymorphism is to allow the functions to have different behavior for each instance. This ad-hoc-ness is constrained by the types in the type class that defines the methods and Haskell's requirement that type class instances be unique for a given type. For any given combination of a

type class and a type, such as `Ord` and `Bool`, there must only be one unique instance in scope. This makes it considerably easier to reason about type classes. See the following example for a disambiguation:

```
(+) :: Num a => a -> a -> a
```

The `+` function is using ad-hoc polymorphism to require the `Num` type class.

```
c' :: a -> a -> a
```

This function is not ad-hoc polymorphic or constrained. It's parametrically polymorphic in the variable `a`.

7. A *module* is the unit of organization that the Haskell programming language uses to collect together declarations of values, functions, datatypes, type classes, and type class instances. Any time you use `import` in Haskell, you are importing declarations from a *module*. Let us look at an example from the chapter exercises:

```
{-# LANGUAGE NoMonomorphismRestriction #-}

module DetermineTheType where
--     ^ name of our module
```

Here, we turn our Haskell source file into a module, and we name it `DetermineTheType`. We include a directive to the compiler to disable the monomorphism restriction before we declare the module. Also, consider the following example using `import`:

```
import Data.Aeson (encode)
--     ^ the module Data.Aeson
import Database.Persist
--     ^ the module Database.Persist
```

In the above example, we are importing the function `encode` declared in the module `Data.Aeson` along with any type class instances. With the module `Database.Persist`, we are importing *everything* it makes available.

5.10 Follow-up resources

1. Luis Damas and Robin Milner. *Principal type-schemes for functional programs*.
   https://web.cs.wpi.edu/~cs4536/c12/milner-damas_principal_types.pdf

2. Christopher Strachey. *Fundamental Concepts in Programming Languages*. The best-known origin of the parametric/ad-hoc polymorphism distinction.
   http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.332.3161&rep=rep1&type=pdf.

Chapter 6

Type Classes

A blank cheque kills creativity.

Mokokoma Mokhonoana

6.1 Type classes

You may have realized that it is very difficult to talk about or understand Haskell's type system without also talking about type classes. So far, we've been focused on the way they interact with type variables and numeric types, especially. This chapter explains some important, predefined type classes, only some of which have to do with numbers, and provides more detail about how type classes work, more generally. In this chapter, we will:

- Examine the type classes `Eq`, `Num`, `Ord`, `Enum`, and `Show`.
- Learn about type-defaulting type classes and how type class inheritance works.
- Look at some common but often implicit functions that create side effects.

6.2 What are type classes?

Type classes and types in Haskell are, in a sense, opposites. Whereas a declaration of a type defines how that type in particular is created, a declaration of a type class defines how a set of types are *consumed* or used in computations. This tension is related to the expression problem, which is about defining code in terms of how data is created or processed. As Philip Wadler put it, "The goal is to define a datatype by cases, where one can add new cases to the datatype and new functions over the datatype, without recompiling existing code, and while retaining static type safety (e.g., no casts)."[1] If you know other programming languages with a similar concept, it may help to think of type classes as being like *interfaces* to data that can work across multiple datatypes. The latter facility is why type classes are a means of ad hoc polymorphism—*ad hoc* because type class code is dispatched by type, something we will explain later in this chapter. We will continue calling it constrained polymorphism, though, as we think that term is generally clearer.

Type classes allow us to generalize over a set of types in order to define and execute a standard set of features for those types. For

[1] See Philip Wadler, *The Expression Problem*: `http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt`.

example, the ability to test values for equality is useful, and we'd want to have it for data of various types. In fact, we can test any data of a type that implements the type class known as `Eq` for equality. We do not need separate equality functions for each different type of data; as long as our datatype implements, or instantiates, the `Eq` type class, we can use the standard functions `==` and `/=`. Similarly, all the numeric literals and their various types implement a type class called `Num`, which defines a standard set of operators that can be used with any type of numbers.

We'll get into more detail about what it means for a type to have an "instance" of a type class in this chapter, but briefly stated, it means that there is code that defines how the values and functions from that type class work for that type. When you use a type class method with one of the types that has such an instance, the compiler looks up the code that dictates how the function works for that type. We'll see this more as we write our own instances.

6.3 Back to Bool

Let's return briefly to the `Bool` type to get a feel for what type class information looks like. As you may recall, we can use the GHCi command `:info` to query information, including type class information about any function or type (and some values):

```
Prelude> :info Bool
data Bool = False | True
instance Eq Bool
instance Ord Bool
instance Show Bool
instance Read Bool
instance Enum Bool
instance Bounded Bool
```

The information includes the data declaration for `Bool` and which type classes it already has instances of. It also tells you where the datatype and its instances are defined for the compiler, if you want to look at the source code, but we've left that information out.

Let's look at that list of instances. Each one is a type class that `Bool` implements, and the instances are the unique specifications of how

`Bool` makes use of the methods from that type class. In this chapter, we're only going to examine a few of these, namely `Eq`, `Ord`, and `Show`. Briefly, however, they mean the following:

1. `instance Bounded Bool`: `Bounded` for types that have an upper and a lower bound.
2. `instance Enum Bool`: `Enum` for things that can be enumerated.
3. `instance Eq Bool`: `Eq` for things that can be tested for equality.
4. `instance Ord Bool`: `Ord` for things that can be put into a sequential order.
5. `instance Read Bool`: `Read` parses strings into things. Don't use it. No, seriously, don't.
6. `instance Show Bool`: `Show` renders things into strings.

Type classes have a hierarchy of sorts,[2] as you might recall from our discussion of numeric types. All `Fractional` numbers implement the `Num` type class, but not all members of `Num` are `Fractional`. All members of `Ord` must be members of `Eq`, and all members of `Enum` must be members of `Ord`. To be able to put things into an enumerated list, they must be orderable; to be orderable, they must be able to be compared for equality.

6.4 Eq

In Haskell, equality is implemented with a type class called `Eq`. Some programming languages bake equality into every object in the language, but some datatypes do not have a sensible notion of equality,[3] so Haskell does not encode equality into every type. `Eq` allows us to use standard measures of equality for quite a few datatypes, though.

`Eq` is defined this way:

[2] You can use a search engine like Hoogle (`http://hoogle.haskell.org`) to find information on Haskell datatypes and type classes. Hoogle is a Haskell API search engine that allows you to search many standard Haskell libraries by function name or type signature. As you become fluent in Haskell types, you will be able to input the type of the function you want in order to find the functions that match that type.

[3] Most importantly, the function type does not have an `Eq` instance for reasons we will not get into here.

```
Prelude> :info Eq
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

First, GHCi tells us we have a type class called `Eq` that specifies two basic functions, equality and non-equality, and gives their type signatures. Next, it prints out all the instances of `Eq` that it knows about. Here is an abbreviated list of the instances you are likely to see in your REPL:

```
instance Eq a => Eq [a]
instance Eq Word
instance Eq Ordering
instance Eq Int
instance Eq Float
instance Eq Double
instance Eq Char
instance Eq Bool
instance (Eq a, Eq b) => Eq (a, b)
instance Eq ()
instance Eq a => Eq (Maybe a)
instance (Eq a, Eq b) => Eq (Either a b)
instance Eq Integer
```

We see several numeric types, our old friend `Bool`, `Char` (unsurprising, as we've seen that we can compare characters for equality), and tuples. We know from this that any time we use values of these types, we are implementing the `Eq` type class and therefore have generic functions we can use to test them for equality. Any type that has an instance of this type class must implement the methods of `Eq`.

Here are some examples using this type class:

```
Prelude> 132 == 132
True
Prelude> 132 /= 132
False
Prelude> (1, 2) == (1, 1)
False
```

```
Prelude> (1, 1) == (1, 2)
False
Prelude> "doge" == "doge"
True
Prelude> "doge" == "doggie"
False
```

The types of `==` and `/=` in `Eq` tell us something important about these functions:

```
(==) :: Eq a => a -> a -> Bool
(/=) :: Eq a => a -> a -> Bool
```

Given these types, we know that they can be used for any type `a` that implements the `Eq` type class. We also know that both functions will take two arguments of the same type `a` and return a value of type `Bool`. We know they have to be the same because `a` must equal `a` in the same type signature.

When we apply `(==)` to a single argument, we can see how it specializes the arguments:

```
(==)
  :: Eq a => a -> a -> Bool
-- if we specialize (==)
-- for [Char] aka String:
(==)
  :: [Char] -> [Char] -> Bool
(==) "cat"
  ::           [Char] -> Bool
(==) "cat" "cat"
  ::                     Bool
```

You can experiment with this further in the REPL to see how applying types to arguments makes the type variables more specific.

What happens if the two arguments `a` and `a` aren't the same type?

```
Prelude> (1, 2) == "puppies!"

• Couldn't match expected type
    ‘(Integer, Integer)’
```

```
              with actual type ‘[Char]’
• In the second argument of ‘(==)’,
    namely ‘"puppies!"’
  In the expression: (1, 2) == "puppies!"
  In an equation for ‘it’:
    it = (1, 2) == "puppies!"
```

Let's break down this type error:

```
• Couldn't match expected type
    ‘(Integer, Integer)’
              with actual type ‘[Char]’
```

This error means our `[Char]` isn't the tuple of `Integer` types that is expected. `(Integer, Integer)` is expected for the second argument (where we supply `"puppies!"`), because that's the type of the first argument. Remember: the type of `a` is usually set by the leftmost occurrence and can't change in the signature `Eq a => a -> a -> Bool`.

Applying `(==)` to `Integer` will bind the `a` type variable to `Integer`, as if the type signature changed to:

```
Eq Integer => Integer -> Integer -> Bool
```

The type class constraint `Eq Integer` gets dropped, because it's redundant. We can see the issue more clearly if we look at the type class instances on the 2-tuple `(,)`:

```
Prelude> :info (,)
data (,) a b = (,) a b
instance Monoid a => Applicative ((,) a)
instance (Eq a, Eq b) => Eq (a, b)
instance Functor ((,) a)
instance Monoid a => Monad ((,) a)
instance (Monoid a, Monoid b)
  => Monoid (a, b)
instance (Ord a, Ord b) => Ord (a, b)
instance (Semigroup a, Semigroup b)
  => Semigroup (a, b)
instance (Show a, Show b) => Show (a, b)
instance (Read a, Read b) => Read (a, b)
```

```
instance Foldable ((,) a)
instance Traversable ((,) a)
instance (Bounded a, Bounded b)
  => Bounded (a, b)
```

We saw the `Eq` instance of `(,)` getting used earlier when we tested code like `(1, 2) == (1, 2)`. Critically, the `Eq` instance of `(a, b)` relies on the `Eq` instances of `a` and `b`. This tells us that the equality of two tuples `(a, b)` depends on the equality of their constituent values `a` and `b`. That is why this works:

```
Prelude> (1, 'a') == (2, 'b')
False
```

But neither of these will work:

```
Prelude> (1, 2) == ('a', 'b')

Prelude> (1, 'a') == ('a', 1)
```

Type class deriving Type class instances we can derive magically include `Eq`, `Ord`, `Enum`, `Bounded`, `Read`, and `Show`, although there are some constraints on deriving a few of these. Deriving means you don't have to manually write instances of these type classes for each new datatype you create. We'll address this a bit more in Chapter 11, on algebraic datatypes.

6.5 Writing type class instances

We haven't talked much about writing your own datatypes yet, or about writing your own type classes. However, you can and will do both. In either case, you will sometimes find yourself needing to write your own type class instances. While `Eq` is one of the type classes you can simply derive, it's also one of the least complicated type classes to write instances for, so we're going to use it here to demonstrate how to write your own instances.

Eq instances

As we've seen, `Eq` provides instances for determining equality of values, so making an instance of it for a given datatype is usually straightforward.

You can investigate a type class by referring to the Hackage documentation for that type class. Type classes like `Eq` come with the core `base` library that is located at `http://hackage.haskell.org/package/base`. `Eq` specifically is located at `http://hackage.haskell.org/package/base/docs/Data-Eq.html`.

In that documentation, you'll want to note a particular bit of wording:

```
Minimal complete definition:
  either == or /=.
```

This tells you what methods you need to define to have a valid `Eq` instance. In this case, either equality, `==`, or inequality, `/=`, will suffice, as one can be defined as the negation of the other. Why not only `==`? Although it's rare, you may have something clever to do for each case that could make equality checking faster for a particular datatype, so you're allowed to specify both if you want to. We won't do that here, because `/=` is the negation of `==`, and we won't be working with any clever datatypes.

First, we'll work with a tiny, trivial datatype called... `Trivial`!

```
data Trivial =
  Trivial
```

With no `deriving` clause hanging off the end of this datatype declaration, we'll have no type class instances of any kind. If we try to load this up and test equality without adding anything further, GHC will throw a type error:

```
Prelude> Trivial == Trivial

• No instance for (Eq Trivial) arising from
    a use of ‘==’
• In the expression: Trivial == Trivial
  In an equation for ‘it’:
    it = Trivial == Trivial
```

GHC can't find an instance of `Eq` for our datatype `Trivial`. We could've had GHC generate one for us using `deriving Eq`, or we could've written one, but we did neither, so none exists, and it fails at compile time. In some languages, this sort of mistake doesn't become known until your code is already in the middle of executing.

Unlike other languages, Haskell does not provide universal stringification or value equality, as this is not always sound or safe, regardless of what programming language you're using.

So we must write our own! Fortunately, with `Trivial` this is... trivial. Keep your type class instances for a type in the same file as that type (we'll explain why later):

```
data Trivial =
  Trivial'

instance Eq Trivial where
  Trivial' == Trivial' = True
```

And that's it! We wrote an instance that tells the compiler how to test this datatype for equality. Data constructors and type constructors often have the same name in Haskell, and that can get confusing. We use the single quote at the end of the data constructor here, because they don't have to have the same name, and it might make it easier to follow the examples.

If you load this up, you have only one possible expression you can construct here:

```
Prelude> Trivial' == Trivial'
True
```

Let's drill down a bit into how this instance stuff works:

```
  instance  Eq   Trivial  where
--  [1]     [2]  [3]      [4]
    Trivial'  ==   Trivial' = True
--    [5]    [6]  [7]        [8]

instance Eq Trivial where
    (==) Trivial' Trivial' = True
--  [          9             ]
```

1. The keyword `instance` here begins a declaration of a type class instance. Type class instances are how you tell Haskell how equality, stringification (`Show`), orderability (`Ord`), enumeration (`Enum`), or other type classes should work for a particular datatype. Without this instance, we can't test the values for equality even though the answer will never vary in the case of this particular datatype.

2. The first name to follow the `instance` is the type class the instance is providing. Here, that is `Eq`.

3. The type the instance is being provided for. In this case, we're implementing the `Eq` type class *for* the `Trivial` datatype.

4. The keyword `where` terminates the initial declaration and beginning of the instance. What follows are the methods (functions) being implemented.

5. The data constructor (value) `Trivial'` is the first argument to the `==` function we're providing. Here, we're defining `==` using infix notation, so the first argument is to the left.

6. The infix function `==`, which is what we're defining in this declaration.

7. The second argument, which is the value `Trivial'`. Since `==` is infix here, the second argument is to the right of the `==`.

8. The result of `Trivial' == Trivial'`, that is, `True`.

9. We could've written the definition of `==` using prefix notation instead of infix by wrapping the operator in parentheses. Note, this is being shown as an alternative; you can't have two type class instances for the same type. Type class instances are unique for a given type. You can try having both in the same file, but you'll get an error.

OK, let's stretch our legs a bit and try something a bit less `Trivial`! We'll make our own datatypes—one for the days of the week, which we'll call `DayOfWeek`, and one for the date that makes use of it:

```
data DayOfWeek =
  Mon | Tue | Weds | Thu | Fri | Sat | Sun

-- day of week and numerical day of month
data Date =
  Date DayOfWeek Int
```

Since these are not pre-baked datatypes in Haskell, they have no type class instances at all. As they stand, there is nothing you can do with them, because no operations are defined for them. Let's fix that. The first `Eq` instance we'll write is for `DayOfWeek`, and you'll see that it is a bit tedious to write out:

```
instance Eq DayOfWeek where
  (==) Mon Mon   = True
  (==) Tue Tue   = True
  (==) Weds Weds = True
  (==) Thu Thu   = True
  (==) Fri Fri   = True
  (==) Sat Sat   = True
  (==) Sun Sun   = True
  (==) _ _       = False
```

Now, we'll write an `Eq` instance for our `Date` type. This one is more interesting:

```
instance Eq Date where
  (==) (Date weekday dayOfMonth)
       (Date weekday' dayOfMonth') =
    weekday == weekday'
  && dayOfMonth == dayOfMonth'
```

In the `Eq` instance for `Date`, we don't recapitulate how equality for `DayOfWeek` and `Int` values works. We simply specify that the dates are equal if all of their constituent values are equal. Note, also, that the compiler already expects the arguments of `Date` to be a `DayOfWeek` value and an `Int`, so we do not need to specify that. Based on what it knows about those three types, this is enough information for us to test `Date` values for equality.

Does it work?

```
Prelude> Date Thu 10 == Date Thu 10
True
Prelude> Date Thu 10 == Date Thu 11
False
Prelude> Date Thu 10 == Date Weds 10
False
```

It compiles, and it returns what we want after three cursory checks—ship it!

We'll point out one other thing about these types:

```
Prelude> Date Thu 10

• No instance for (Show Date) arising from
    a use of ‘print’
• In a stmt of an interactive GHCi command:
    print it
```

We wrote an `Eq` instance, so we can test the values for equality, but we can't print them in the REPL, because we don't provide a `Show` instance. If you'd like to fix that, you can stick a `deriving Show` clause on the end of each of the datatypes above.

Partial functions—not so strange danger

We've mentioned partial application of functions previously, but the term *partial function* refers to something different. A partial function is one that doesn't handle all the possible input cases, so there are scenarios in which we haven't defined any way for the code to evaluate.

We need to take care to avoid partial functions in general in Haskell, but this must be kept in mind especially when we have a type with multiple cases such as `DayOfWeek`. What if we had made a mistake in the `Eq` instance?

```
data DayOfWeek =
  Mon | Tue | Weds | Thu | Fri | Sat | Sun

instance Eq DayOfWeek where
  (==) Mon Mon   = True
  (==) Tue Tue   = True
  (==) Weds Weds = True
  (==) Thu Thu   = True
  (==) Fri Fri   = True
  (==) Sat Sat   = True
  (==) Sun Sun   = True
```

What if the arguments are different? We forgot our unconditional case. This will appear to be fine whenever the arguments are the same, but blow up in our faces when they're not:

```
Prelude> Mon == Mon
True

Prelude> Mon == Tue
*** Exception: code/derivingInstances.hs:
(19,3)-(25,23):
  Non-exhaustive patterns in function ==
```

Well, that stinks. We definitely didn't start learning Haskell because we want stuff to blow up at runtime. So what gives?

The good news is there *is* something you can do to get more help from GHC on this. If we turn all warnings on with the `-Wall` flag in our REPL (or in our build configuration), then GHC will let us know when we're not handling all cases:

```
Prelude> :set -Wall
Prelude> :l code/derivingInstances.hs
[1 of 1] Compiling DerivingInstances

    Pattern match(es) are non-exhaustive
    In an equation for ‘==’:
        Patterns not matched:
            Mon Tue
```

```
            Mon Weds
            Mon Thu
            Mon Fri
            ...
Ok, one module loaded.
```

You'll find that if you fix your instance and provide the fallback case that returns `False`, it'll stop squawking about the non-exhaustive patterns.

Partial functions are not only a concern with type class instances, though. We will discuss this more in the next chapter, but it's also a concern with any function that doesn't handle all possible inputs, such as this one, which blows up anytime the input isn't 2:

```
f :: Int -> Bool
f 2 = True
```

If you compile or load this code, you'll get another warning (assuming you still have `-Wall` turned on). In this case, because `Int` is a *huge* type with many values, it's using notation that says you're not handling all inputs that aren't the number 2:

```
    Pattern match(es) are non-exhaustive
    In an equation for ‘f’:
        Patterns not matched:
          p where p is not one of {2}
```

If you add another case such that you're handling one more input, it will add that to the set of values you are handling:

```
f :: Int -> Bool
f 1 = True
f 2 = True
```

```
    Pattern match(es) are non-exhaustive
    In an equation for ‘f’:
        Patterns not matched:
          p where p is not one of {2, 1}
```

```
f :: Int -> Bool
f 1 = True
f 2 = True
f 3 = True
```

```
    Pattern match(es) are non-exhaustive
    In an equation for ‘f’:
        Patterns not matched:
          p where p is not one of {3, 2, 1}
```

So on and so forth. The real answer here is to have an unconditional case that matches everything. The following will compile without complaint and is not partial:

```
f :: Int -> Bool
f 1 = True
f 2 = True
f 3 = True
f _ = False
```

Another solution is to use a datatype that isn't *huge* like `Int`, if you only have a few cases you want to consider:

```
Prelude> minBound :: Int
-9223372036854775808
Prelude> maxBound :: Int
9223372036854775807
```

Seriously. It's huge.

If you want your data to describe only a handful of cases, write them down in a sum type like the `DayOfWeek` datatype we showed you earlier. Don't use `Int` as an implicit sum type as C programmers commonly do.

Sometimes we need to ask for more

When we're writing an instance of a type class such as `Eq` for something with polymorphic parameters, such as `Identity` below, we'll sometimes need to require our argument or arguments to provide some type class instances for us in order to write an instance for the datatype containing them:

```
data Identity a =
  Identity a

instance Eq (Identity a) where
  (==) (Identity v) (Identity v') = v == v'
```

What we want to do here is rely on whatever `Eq` instances the argument to `Identity` (`a` in the datatype declaration and `v` in the instance definition, above) has already. There is one problem with this as it stands, though:

```
• No instance for (Eq a) arising from a use
    of ‘==’
  Possible fix: add (Eq a) to the context
    of the instance declaration
• In the expression: v == v'
  In an equation for ‘==’:
    (==) (Identity v) (Identity v') =
      v == v'
  In the instance declaration for
    ‘Eq (Identity a)’
```

The problem here is that `v` and `v'` are both of type `a`, but we don't know anything about `a`. In this case, we can't assume it has an `Eq` instance. However, we can use the same type class constraint syntax we saw with functions in our instance declaration:

```
instance Eq a => Eq (Identity a) where
  (==) (Identity v) (Identity v') = v == v'
```

Now it'll work, because we know `a` has to have an instance of `Eq`. Additionally, Haskell will ensure we don't attempt to check equality with values that don't have an `Eq` instance at compile time:

```
data NoEq = NoEqInst deriving Show

Prelude> inoe = Identity NoEqInst
Prelude> inoe == inoe

• No instance for (Eq (Identity NoEq))
```

```
    arising from a use of '=='
• In the expression: inoe == inoe
  In an equation for 'it':
    it = inoe == inoe
```

We could ask for more than we need in order to obtain an answer, such as below where we ask for an `Ord` instance for `a`, but there's no reason to do so, since `Eq` requires less than `Ord` and does enough for what we need here:

```
instance Ord a => Eq (Identity a) where
  (==) (Identity v) (Identity v') =
    compare v v' == EQ
```

This will compile, but it's not clear why you'd do it. Maybe you have your own secret reasons.

Exercises: Eq instances

Write the `Eq` instances for the datatypes provided.

1. It's not a typo, we're just being cute with the name:

   ```
   data TisAnInteger =
     TisAn Integer
   ```

2. ```
   data TwoIntegers =
     Two Integer Integer
   ```

3. ```
   data StringOrInt =
     TisAnInt   Int
   | TisAString String
   ```

4. ```
   data Pair a =
     Pair a a
   ```

5. ```
   data Tuple a b =
     Tuple a b
   ```

6. ```
   data Which a =
     ThisOne a
   | ThatOne a
   ```
   ```

7.
```
data EitherOr a b =
       Hello a
     | Goodbye b
```

6.6 Num

We have seen a lot of `Num` at this point, so we'll try not to go on too long about it. It is a type class implemented by most numeric types. As we did with `Eq`, we will query the information and examine its set of predefined functions:

```
class Num a where
  (+) :: a -> a -> a
  (*) :: a -> a -> a
  (-) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a
```

And its list of instances (not quite complete):

```
instance Num Integer
instance Num Int
instance Num Float
instance Num Double
```

We've seen most of this information before, in one form or another: common arithmetic functions with their type signatures at the top (`fromInteger` is similar to `fromIntegral` but restricted to `Integer` rather than all integral numbers) plus a list of types that implement this type class, numeric types we've looked at previously. No surprises here.

Integral

The type class called `Integral` has the following definition:

```
class (Real a, Enum a) => Integral a where
  quot :: a -> a -> a
  rem :: a -> a -> a
  div :: a -> a -> a
  mod :: a -> a -> a
  quotRem :: a -> a -> (a, a)
  divMod :: a -> a -> (a, a)
  toInteger :: a -> Integer
```

The type class constraint `(Real a, Enum a)` means that any type that implements `Integral` must already have instances for `Real` *and* `Enum` type classes. In a very real sense, the tuple syntax here denotes the conjunction of type class constraints on your type variables. An integral type must be both a real number and enumerable and therefore may employ the methods of each of those type classes. In turn, the `Real` type class itself requires an instance of `Num`. So, the `Integral` type class may put the methods of `Real` and `Num` into effect (in addition to those of `Enum`). Since `Real` cannot override the methods of `Num`, this type class inheritance is *only* additive, and therefore Haskell avoids the ambiguity problems—the so-called "deadly diamond of death"—caused by multiple inheritance in some programming languages.

Exercise: Tuple experiment Look at the types given for `quotRem` and `divMod`. What do you think those functions do? Test your hypotheses by playing with them in the REPL. We've given you a sample to start with below:

```
Prelude> ones x = snd (divMod x 10)
```

Fractional

`Num` is a superclass of `Fractional`. The `Fractional` type class is defined as follows:

```
class (Num a) => Fractional a where
    (/)          :: a -> a -> a
    recip        :: a -> a
    fromRational :: Rational -> a
```

This type class declaration creates a class named `Fractional`, which requires its type argument `a` to have an instance of `Num` in order to create an instance of `Fractional`. This is another example of type class inheritance. `Fractional` applies to fewer numbers than `Num` does, and instances of the `Fractional` class can use the functions defined in `Num`, but not all `Num` values can use the functions defined in `Fractional`, because nothing in `Num`'s definition requires an instance of `Fractional`. There is a chart at the end of this chapter to help you visualize this information.

We can see this with ordinary functions. First, let's consider this function, intentionally without a type provided:

```
divideThenAdd x y = (x / y) + 1
```

We'll load this with a type that asks only for a `Num` instance:

```
divideThenAdd :: Num a => a -> a -> a
divideThenAdd x y = (x / y) + 1
```

And you'll get the type error:

```
• Could not deduce (Fractional a) arising
    from a use of ‘/’
  from the context: Num a
    bound by the type signature for:
           divideThenAdd ::
           forall a. Num a => a -> a -> a
```

Now, if we only cared about having the `Num` constraint, we could modify our function to not use `/`, which requires `Fractional`:

```
subtractThenAdd :: Num a => a -> a -> a
subtractThenAdd x y = (x - y) + 1
```

This works fine. `+` and `-` are both provided by `Num`. Or we can change the type rather than modifying the function itself:

```
divideThenAdd :: Fractional a
              => a -> a -> a
divideThenAdd x y = (x / y) + 1
```

This also works fine.

Put on your thinking cap Why didn't we need to make the type of the function we wrote require both type classes? Why didn't we have to do this:

```
f :: (Num a, Fractional a) => a -> a -> a
```

Consider what it means for something to be a *subset* of a larger set of objects.

6.7 Type-defaulting type classes?

When you have a type class-constrained (ad hoc), polymorphic value and need to evaluate it, the polymorphism must be resolved to a specific concrete type. The concrete type must have an instance for all the required type class instances (that is, if it is required to implement `Num` *and* `Fractional`, then the concrete type can't be an `Int`). Ordinarily, the concrete type would come from the type signature you've specified or from type inference, such as when a `Num a => a` is used in an expression that expects an `Integer`, which forces the polymorphic number value to concretize as an `Integer`. But in some cases, particularly when you're working in the GHCi REPL, you will not have specified a concrete type for a polymorphic value. In those situations, the type class will default to a concrete type, and the default types are already set in the libraries.

When we do this in the REPL:

```
Prelude> 1 / 2
0.5
```

Our result `0.5` appears the way it does, because it defaults to `Double`. Using the type assignment operator `::`, we can assign a more specific type and circumvent the default to `Double`:

```
Prelude> 1 / 2 :: Float
0.5
Prelude> 1 / 2 :: Double
0.5
Prelude> 1 / 2 :: Rational
1 % 2
```

The Haskell Report[4] specifies the following defaults relevant to numerical computations:

```
default Num Integer
default Real Integer
default Enum Integer
default Integral Integer
default Fractional Double
default RealFrac Double
default Floating Double
default RealFloat Double
```

`Num`, `Real`, etc., are type classes, and `Integer` and `Double` are the types they default to. This type defaulting for `Fractional` means that:

```
(/) :: Fractional a => a -> a -> a
```

Changes to:

```
(/) :: Double -> Double -> Double
```

If you don't specify the concrete type desired for `/`. A similar example, but for `Integral`, would be:

```
div :: Integral a => a -> a -> a
```

Defaulting to:

```
div :: Integer -> Integer -> Integer
```

The type class constraint is superfluous when the types are concrete. On the other hand, you must specify which type classes you want your type variables to implement. The use of polymorphic values without the ability to infer a specific type and no default rule will cause GHC to complain about ambiguous types.

The following examples will work, because all the types below implement the `Num` type class:

[4]The Haskell Report is the document that specifies the language and standard libraries for Haskell. The most recent version is Haskell Report 2010, which can be found at `https://www.haskell.org/onlinereport/haskell2010/`.

```
Prelude> x = 5 + 5 :: Int
Prelude> x
10

Prelude> x = 5 + 5 :: Integer
Prelude> x
10

Prelude> x = 5 + 5 :: Float
Prelude> x
10.0

Prelude> x = 5 + 5 :: Double
Prelude> x
10.0
```

Now, we can make this type more specific, and the process will be similar. In this case, let's use `Integer`, which implements `Num`:

```
let x = 10 :: Integer
let y = 5 :: Integer
```

These are the declared types for these functions, because they're from `Num`:

```
(+)   :: Num a => a -> a -> a
(*)   :: Num a => a -> a -> a
(-)   :: Num a => a -> a -> a
```

Any functions from `Num` are going to automatically get specialized to `Integer` when we apply them to the `x` or `y` values:

```
Prelude> :t (x+)
(x+) :: Integer -> Integer

-- For
(+)   :: Num a => a -> a -> a
-- When 'a' is type Integer
(+)   :: Integer -> Integer -> Integer
-- Apply the first argument
```

```
(x+)  ::            Integer -> Integer
-- Apply the second and last argument
(x+y) ::                       Integer
-- Final result is Integer
```

We can declare more specific (monomorphic) functions from more general (polymorphic) functions:

```
add = (+) :: Integer -> Integer -> Integer
```

We cannot go in the other direction, because we lose the generality of `Num` when we specialize to `Integer`:

```
Prelude> :t id
id :: a -> a
Prelude> numId = id :: Num a => a -> a
Prelude> intId = numId :: Int -> Int
Prelude> numId' = intId :: Num a => a -> a

• Couldn't match type ‘a1’ with ‘Int’
  ‘a1’ is a rigid type variable bound by
    an expression type signature:
      forall a1. Num a1 => a1 -> a1

  Expected type: a1 -> a1
    Actual type: Int -> Int
• In the expression:
    intId :: Num a => a -> a
  In an equation for ‘numId'’:
    numId' = intId :: Num a => a -> a
```

The *expected type* and the *actual type* don't match. Remember, the actual type is the type we provide; the expected type is what the compiler expects. Here, the actual type is more concrete than the expected type. Types can be made more specific, but they cannot be made more general or polymorphic.

6.8 Ord

Next, we'll take a look at a type class called `Ord`. We previously noted that this type class covers the types of things that can be put in order.

If you use `:info` for `Ord` in your REPL, you will find a very large number of instances for this type class. We're going to pare it down a bit and focus on the essentials, but, as always, we encourage you to explore this further on your own:

```
Prelude> :info Ord
class Eq a => Ord a where
  compare :: a -> a -> Ordering
  (<) :: a -> a -> Bool
  (>=) :: a -> a -> Bool
  (>) :: a -> a -> Bool
  (<=) :: a -> a -> Bool
  max :: a -> a -> a
  min :: a -> a -> a

instance (Ord a, Ord b) => Ord (Either a b)
instance Ord a => Ord (Maybe a)
instance Ord a => Ord [a]
instance Ord Word
instance Ord Ordering
instance Ord Int
instance Ord Float
instance Ord Double
instance Ord Char
instance Ord Bool
instance (Ord a, Ord b) => Ord (a, b)
instance Ord ()
instance Ord Integer
```

Notably, at the top, we have another type class constraint. `Ord` is constrained by `Eq`, because if you're going to compare items in a list and put them in order, you need a way to determine if they are equal. So, `Ord` requires `Eq` and its methods. The functions that come standard in this class have to do with ordering. Some of them will give you a result of `Bool`, and we've played a bit with those functions. Let's see what a few others do:

```
Prelude> compare 7 8
LT
```

```
Prelude> compare 4 (-4)
GT
Prelude> compare 4 4
EQ
Prelude> compare "Julie" "Chris"
GT
Prelude> compare True False
GT
Prelude> compare True True
EQ
```

The `compare` function works for any of the types listed above that implement the `Ord` type class, including `Bool`, but unlike the `<`, `>`, `>=`, and `<=` operators, this returns an `Ordering` value instead of a `Bool` value.

You may notice that `True` is greater than `False`. Proximally, this is due to how the `Bool` datatype is defined: `False | True`. There may be a more interesting underlying reason if you prefer to ponder the philosophical implications.

The `max` and `min` functions work in a similarly straightforward fashion for any type that implements this type class:

```
Prelude> max 7 8
8
Prelude> min 10 (-10)
-10
Prelude> max (3, 4) (2, 3)
(3,4)
Prelude> min [2, 3, 4, 5] [3, 4, 5, 6]
[2,3,4,5]
Prelude> max "Julie" "Chris"
"Julie"
```

By looking at their type signatures, we can see that these functions have two parameters. If you want to use them to determine the maximum or minimum of three values, you can nest them:

```
Prelude> max 7 (max 8 9)
9
```

If you try to give it too few arguments, you will get this strange-seeming message, which we'll break down below:

```
• No instance for (Show ([Char] -> [Char]))
 --      [1]        [2]   [       3      ]
     arising from a use of ‘print’
 --                          [4]
    (maybe you haven't applied a function
     to enough arguments?)
• In a stmt of an interactive GHCi command:
    print it
--  [  5  ]
```

1. Haskell can't find an instance of a type class for a value of a given type.
2. The type class it can't find an instance for is `Show`, the type class that allows GHCi to print values in your terminal. More on this in the following sections.
3. It can't find an instance of `Show` for the type `String -> String`. Nothing with the `->` type should have a `Show` instance as a general rule, because `->` denotes a function rather than a constant value.
4. We want an instance of `Show`, because we (indirectly) invok `print`, which has type `print :: Show a => a -> IO ()`—note the constraint for `Show`.
5. The interactive GHCi command `print it` invokes `print` on our behalf.

Any time we ask GHCi to print a return value in our terminal, we are indirectly invoking `print`, which has the type `Show a => a -> IO ()`. The first argument to `print` must have an instance of `Show`. The error message appears, because `max` applied to a single `String` argument needs another argument before it'll return a `String` (aka `[Char]`) value that is `Show`-able or printable. Until we apply it to a second argument, it's still a function, and a function has no instance of `Show`. The request to `print` a function, rather than a constant value, results in this error message.

Ord instances

We'll see more examples of writing instances as we proceed in the book and explain more thoroughly how to write your own datatypes. We wrote some `Eq` instances earlier. Now, we'll practice our instance-writing skills (this is one of the most necessary skills in Haskell) by writing `Ord` instances.

When you derive `Ord` instances for a datatype, they rely on the way the datatype is defined, but if you write your own instance, you can define the behavior you want. We'll use the days of the week again to demonstrate:

```
data DayOfWeek =
  Mon | Tue | Weds | Thu | Fri | Sat | Sun
  deriving (Ord, Show)
```

We only derive `Ord` and `Show` there, because you should still have the `Eq` instance we wrote for this datatype in scope. If you don't, you have two options: bring it back into scope by putting it into the file you're currently using, or derive an `Eq` instance for the datatype now by adding it inside the parentheses. You can't have an `Ord` instance unless you also have an `Eq` instance, so the compiler will complain if you don't do one (not both) of those two things.

Values to the left are *less than* values to the right, as if they were placed on a number line:

```
Prelude> Mon > Tue
False
Prelude> Sun > Mon
True
Prelude> compare Tue Weds
LT
```

But if we want to express that Friday is always the best day, we can write our own `Ord` instance to express that:

```
data DayOfWeek =
  Mon | Tue | Weds | Thu | Fri | Sat | Sun
  deriving (Eq, Show)

instance Ord DayOfWeek where
  compare Fri Fri = EQ
  compare Fri _   = GT
  compare _ Fri   = LT
  compare _ _     = EQ
```

Now, if we compare Friday to any other day, Friday is always greater. All other days, as you can see, are equal in value:

```
Prelude> compare Fri Sat
GT
Prelude> compare Sat Mon
EQ
Prelude> compare Fri Mon
GT
Prelude> compare Sat Fri
LT
Prelude> Mon > Fri
False
Prelude> Fri > Sat
True
```

But we did derive an `Eq` instance above, so we do get the expected equality behavior:

```
Prelude> Sat == Mon
False
Prelude> Fri == Fri
True
```

A few things to keep in mind about writing `Ord` instances: First, it is wise to ensure that your `Ord` instances agree with your `Eq` instances, whether the `Eq` instances are derived or manually written. If `x == y`, then `compare x y` should return `EQ`. Also, you want your `Ord` instances to define a sensible total order. You ensure this in part by covering

all cases and not writing partial instances, as we noted above with `Eq`. In general, your `Ord` instance should be written such that, when `compare x y` returns `LT`, then `compare y x` returns `GT`.

Ord implies Eq

The following isn't going to type check for reasons we have already covered:

```
check' :: a -> a -> Bool
check' a a' = a == a'
```

The error we get mentions that we need `Eq`, which makes sense!

```
• No instance for (Eq a) arising from a use
    of ‘==’
  Possible fix:
    add (Eq a) to the context of
      the type signature for:
        check' :: forall a. a -> a -> Bool
• In the expression: a == a'
  In an equation for ‘check'’:
    check' a a' = a == a'
```

But what if we add `Ord` instead of `Eq`, as it asks?

```
check' :: Ord a => a -> a -> Bool
check' a a' = a == a'
```

It should compile. Now, `Ord` isn't what GHC asks for, so why does it work? It works, because anything that provides an instance of `Ord` *must* by definition also already have an instance of `Eq`. How do we know? As we said above, logically it makes sense that you can't order things without the ability to check for equality, but we can also check `:info Ord` in GHCi:

```
Prelude> :info Ord
class Eq a => Ord a where

...buncha noise we don't care about...
```

The class definition of `Ord` says that any `a` that wants to define an `Ord` instance must already provide an `Eq` instance. We can say that `Eq` is a *superclass* of `Ord`.

Usually, you want the *minimally sufficient* set of constraints on all your functions—so we would use `Eq` instead of `Ord` if the above example were "real" code—but we did this so you could get an idea of how constraints and superclassing work.

Exercises: Will they work?

Next, take a look at the following code examples, and try to decide if they will work, what result they will return if they do, and why or why not (be sure, as always, to test them in your REPL once you have decided on your answers):

1. ```
   max (length [1, 2, 3])
       (length [8, 9, 10, 11, 12])
   ```

2. ```
   compare (3 * 4) (3 * 5)
   ```

3. ```
   compare "Julie" True
   ```

4. ```
   (5 + 3) > (3 + 6)
   ```

6.9 Enum

A type class known as `Enum` that we have mentioned previously seems similar to `Ord` but is slightly different. This type class covers types that are enumerable, that is, their values have known predecessors and successors. We won't belabor the point, because you are probably developing a good idea of how to query and make use of type class information:

```
Prelude> :info Enum
class Enum a where
  succ :: a -> a
  pred :: a -> a
  toEnum :: Int -> a
  fromEnum :: a -> Int
  enumFrom :: a -> [a]
```

```
  enumFromThen :: a -> a -> [a]
  enumFromTo :: a -> a -> [a]
  enumFromThenTo :: a -> a -> a -> [a]

instance Enum Word
instance Enum Ordering
instance Enum Integer
instance Enum Int
instance Enum Char
instance Enum Bool
instance Enum ()
instance Enum Float
instance Enum Double
```

Numbers and characters have predictable successors and predecessors, so these are paradigmatic cases of enumerability:

```
Prelude> succ 4
5
Prelude> pred 'd'
'c'
Prelude> succ 4.5
5.5
```

You can also see that some of these functions return a result of a list type. They take a starting value and build a list with the succeeding items of the same type:

```
Prelude> enumFromTo 3 8
[3,4,5,6,7,8]
Prelude> enumFromTo 'a' 'f'
"abcdef"
```

Finally, let's take a short look at `enumFromThenTo`:

```
Prelude> enumFromThenTo 1 10 100
[1,10,19,28,37,46,55,64,73,82,91,100]
```

Take a look at the resulting list, and see if you can find the pattern: what does this function do? What happens if we give it the values `0 10 100` instead? How about `'a' 'c' 'z'`?

6.10 Show

`Show` is a type class that provides for the creating of human-readable string representations of structured data. GHCi uses `Show` to generate `String` values it can print in the terminal.

`Show` is not a serialization format. Serialization is how data is rendered to a textual or binary format for persistence or communicating with other computers over a network. An example of persistence would be saving data to a file on disk. `Show` is not suitable for any of these purposes; it's expressly for human readability.

The type class information looks like this (truncated):

```
class Show a where
  showsPrec :: Int -> a -> ShowS
  show :: a -> String
  showList :: [a] -> ShowS

instance (Show a, Show b) =>
  Show (Either a b)
instance Show a => Show [a]
instance Show Word
instance Show Ordering
instance Show a => Show (Maybe a)
instance Show Integer
instance Show Int
instance Show Char
instance Show Bool
instance Show ()
instance Show Float
instance Show Double
```

Importantly, we see that various number types, `Bool` values, tuples, and characters are all already instances of `Show`. That is, they have a defined ability to be printed to the screen. There is also a function `show` that takes a polymorphic `a` and returns it as a `String`, allowing it to be printed.

Printing and side effects

When you ask GHCi to return the result of an expression and print it to the screen, you are indirectly invoking a function called `print` that we encountered briefly in Chapter 3, again in the section about `Ord`, and most recently in the error message that results from passing too few arguments to the `max` function. As understanding `print` is important to understanding this type class, we're going to digress a bit and talk about it in more detail.

Haskell is a pure functional programming language. The *functional* part of that comes from the fact that programs are written as functions, similar to mathematical equations, in which an operation is applied to some arguments to produce a result. The *pure* part of our description of Haskell means expressions in Haskell can be expressed exclusively in terms of a lambda calculus.

It may not seem obvious that printing results to the screen could be a source of worry. The function is not just applied to the arguments that are in its scope but also asked to affect the world outside its scope in some way, namely by showing you its result on a screen. This is known as a *side effect*, a potentially observable result apart from the value the expression evaluates to. Haskell manages effects by separating effectful computations from pure computations in ways that preserve the predictability and safety of function evaluation. Importantly, effect-bearing computations themselves become more composable and easier to reason about. The benefits of explicit effects include the fact that it makes it relatively easy to reason about and predict the results of our functions.

What sets Haskell apart from most other functional programming languages is that it introduced and refined a means of writing ordinary programs that talk to the outside world without adding anything to the pure lambda calculus it is founded on. This property—being lambda calculus and nothing more—is what makes Haskell a purely functional programming language.

The `print` function is sometimes invoked indirectly by GHCi, but its type explicitly reveals that it is effectful. Up to now, we've been glossing over how this works, but it's time to dive a bit deeper.

`print` is defined in the `Prelude` as a function that "outputs a value of any printable type to the standard output device. Printable types are those that are instances of class `Show`; `print` converts values to strings

for output using the `show` operation and adds a newline." Let's look at the type of `print`:

```
Prelude> :t print
print :: Show a => a -> IO ()
```

As we see, `print` takes an argument `a` that must be a type with an instance of the `Show` type class and returns an `IO ()` result. This result is an `IO` action that returns a value of type `()`.

We saw this `IO ()` result previously when we talked about printing strings. We also noted that it is the obligatory type of `main` in a source code file. This is because running `main` *only* produces side effects.

Stated as simply as possible, an I/O[5] action is an action that, when performed, has side effects, including reading from input and printing to the screen, and will contain a return value. The `()` denotes an empty tuple, which we refer to as *unit*. Unit is a value and also a type that has only this one inhabitant, which essentially represents nothing. Printing a string to the terminal doesn't have a meaningful return value. But an IO action, like any expression in Haskell, can't return *nothing*; it must return something. So we use this empty tuple to represent the return value at the end of our IO action. That is, the `print` function will first do the IO action of printing the string to the terminal and then complete the action, marking an end to the execution of the function and a delimitation of the side effects, by returning this empty nothing tuple. It does not print the empty tuple to the screen, but it is implicitly there. The simplest way to think about the difference between a value with a typical type like `String` and the same type but from `IO`, such as `IO String`, is that IO actions are formulas. When you have a value of type `IO String`, it's more of a *means of producing* a `String`, which may require performing side effects along the way before you get your `String` value.

This is a `String` value:

```
myVal :: String
```

This value is a *method* or means of obtaining a value, by performing effects or I/O, of type `String`:

[5] Input/output, frequently written "IO" without a slash; when referring to the Haskell datatype, there is no slash.

```
ioString :: IO String
```

An `IO` action is performed when we call `main` for our program, as we have seen. But we also perform an `IO` action when we invoke `print` implicitly or explicitly.

Working with Show

Up to now, we have only been deriving type class instances for `Show`, because deriving usually gives us the result we want without a lot of fuss. Having a `Show` instance is crucial to being able to print anything to the terminal, so we're going to look at some examples of why `Show` is important and how it is implemented. Invoking the `Show` type class also invokes its methods, specifically a method of taking your values and turning them into values that can be printed to the screen.

A minimal implementation of an instance of `Show` only requires that `show` or `showsPrec` be implemented, as in the following example:

```
data Mood = Blah

instance Show Mood where
  show _ = "Blah"


Prelude> Blah
Blah
```

Here's what happens in GHCi when you define a datatype and ask GHCi to show it without the instance for the `Show` type class:

```
Prelude> data Mood = Blah
Prelude> Blah

• No instance for (Show Mood) arising from
    a use of ‘print’
• In a stmt of an interactive GHCi command:
    print it
```

Next, let's look at how you define a datatype to have an instance of `Show`. We can derive the `Show` instance for `Mood`, because it's one of the type classes for which GHC supports deriving instances by default:

```
Prelude> data Mood = Blah deriving Show
Prelude> Blah
Blah
```

And, in fact, most of the time that's what you'll do for your own datatypes. In Chapter 13, on building projects, we will need to write a custom instance for `Show`, however, so that should give you something exciting to look forward to.

6.11 Read

The `Read` type class... well, it's... *there*. You'll notice that, like `Show`, a lot of types have instances of `Read`. This type class is essentially the opposite of `Show`. Whereas `Show` takes things and turns them into human-readable strings, `Read` takes strings and turns them into things. Like `Show`, it's not a serialization format. So, what's the problem? We gave that dire warning against using `Read` earlier in the chapter, but this doesn't seem like a big deal, right?

The problem is in the `String` type. A `String` is a list, which could be empty in some cases or stretch on to infinity in other cases.

We can begin to understand this by examining the types:

```
Prelude> :t read
read :: Read a => String -> a
```

There's no way `Read a => String -> a` will always work. Let's consider a type like `Integer`, which has a `Read` instance. We have no guarantee that the `String` will be a valid representation of an `Integer` value. A `String` value can be *any* text. That's way too big of a type for things we want to parse into numbers! We can prove this for ourselves in the REPL:

```
Prelude> read "1234567" :: Integer
1234567
Prelude> read "BLAH" :: Integer
*** Exception: Prelude.read: no parse
```

That exception is a runtime error and means that `read` is a *partial function*, a function that doesn't return a proper value as a result *for all possible* inputs. We have ways of cleaning this up we'll explain and

demonstrate later. We should strive to avoid writing or using such functions in Haskell, because Haskell gives us the tools necessary to avoid senseless sources of errors in our code.

6.12 Instances are dispatched by type

We've said a few times, without explaining it, that type classes are dispatched by type, but it's an important thing to understand. Type classes are defined by the set of operations and values that all instances must provide. Type class *instances* are unique pairings of a type class and a type. They define the ways to implement the type class methods for that type.

We're going to walk through some code to illustrate what all this means. The first thing you will see is that we've written our own type class and instances for demonstration purposes. Those details aren't important for understanding this code. Just remember:

- A type class defines a set of functions and/or values.
- Types have instances of that type class.
- The instances specify the ways that type uses the functions of the type class.

The following example is vacuous and silly—it's only to make a point. Please do not write type classes like this:

```
class Numberish a where
  fromNumber :: Integer -> a
  toNumber :: a -> Integer


-- pretend newtype is data for now
newtype Age =
  Age Integer
  deriving (Eq, Show)

instance Numberish Age where
  fromNumber n = Age n
  toNumber (Age n) = n
```

```
newtype Year =
  Year Integer
  deriving (Eq, Show)

instance Numberish Year where
  fromNumber n = Year n
  toNumber (Year n) = n
```

Then, suppose we write a function using this type class and the two types and instances:

```
sumNumberish :: Numberish a => a -> a -> a
sumNumberish a a' = fromNumber summed
  where integerOfA      = toNumber a
        integerOfAPrime = toNumber a'
        summed =
          integerOfA + integerOfAPrime
```

Let us think about this for a moment. The class definition of `Numberish` doesn't define any *terms* or code we can compile and execute, only types. The code lives in the instances for `Age` and `Year`. So how does Haskell know where to find that code?

```
Prelude> sumNumberish (Age 10) (Age 10)
Age 20
```

In the above, it knows to use the instance of `Numberish` for `Age`, because it can see that our arguments to `sumNumberish` are of type `Age`. We can see this with the type inference, too:

```
Prelude> :t sumNumberish
sumNumberish :: Numberish a => a -> a -> a


Prelude> :t sumNumberish (Age 10)
sumNumberish (Age 10) :: Age -> Age
```

After the first parameter is applied to a value of type `Age`, it knows that all other occurrences of type `Numberish a => a` must be `Age`.

To see a case where we're *not* providing enough information to Haskell for it to identify a concrete type with which to get the appropriate instance, we're going to change our type class and associated

instances (this is even worse than the last one. Don't use type classes to define default values. Seriously. Haskell ninjas will find you and replace your toothpaste with muddy chalk):

```
class Numberish a where
  fromNumber    :: Integer -> a
  toNumber      :: a -> Integer
  defaultNumber :: a


instance Numberish Age where
  fromNumber n = Age n
  toNumber (Age n) = n
  defaultNumber = Age 65

instance Numberish Year where
  fromNumber n = Year n
  toNumber (Year n) = n
  defaultNumber = Year 1988
```

Then in the REPL, we can see that in some cases, there's no way for Haskell to know what we want:

```
Prelude> defaultNumber

• Ambiguous type variable ‘a0’ arising from
  a use of ‘print’ prevents the constraint
  ‘(Show a0)’ from being solved.
  Probable fix: use a type annotation to
  specify what ‘a0’ should be.
  These potential instances exist:
    instance Show Ordering
    instance Show Integer
    instance [safe] Show Age
    ...plus 24 others
    ...plus 18 instances involving
      out-of-scope types
    (use -fprint-potential-instances to
      see them all)
```

```
• In a stmt of an interactive GHCi command:
    print it
```

This fails, because it has *no idea* what type `defaultNumber` is other than that it's provided for by the instances of `Numberish`. But the good news is, even if it's a value and doesn't take any arguments, we have a means of telling Haskell what we want:

```
Prelude> defaultNumber :: Age
Age 65
Prelude> defaultNumber :: Year
Year 1988
```

Just assign the type you expect, and it works fine! Here, Haskell is using the type assertion to *dispatch*, or specify, which type class instance we want to get our `defaultNumber` from.

Why not write a type class like this? For reasons we'll explain when we talk about `Monoid`, it's important that your type classes have laws and rules about how they work. `Numberish` is a bit... arbitrary. There are better ways to express what it does in Haskell than a type class. Functions and values alone suffice here.

6.13 Gimme more operations

We talked about the different kinds of polymorphism in type signatures: constrained vs. parametric. Having no constraint on our term-level values means they could be any type, but there isn't much we can do with them. The methods and operations are in the type classes, and so we get more utility by specifying type class constraints. If your types are more general than your terms are, then you need to constrain your types with the type classes that provide the operations you want to use. We looked at some examples of this in the sections above about `Integral` and `Fractional`, but in this section, we'll be more specific about how to modify type signatures to fit the terms.

We'll start by looking at some examples of places where we need to change our types, because they're more general than our terms allow:

```
add :: a -> a -> a
add x y = x + y
```

If you load it up, you'll get the following error:

```
• No instance for (Num a) arising from a
    use of ‘+’
  Possible fix:
    add (Num a) to the context of
      the type signature for:
        add :: forall a. a -> a -> a
• In the expression: x + y
  In an equation for ‘add’: add x y = x + y
```

Fortunately, this is one of those cases where GHC knows precisely what the problem is and how to remedy it. We need to add a `Num` constraint to the type `a`. But why? Because our function can't accept a value of strictly *any* type. We need something that has an instance of `Num`, because the `+` function comes from `Num`:

```
add :: Num a => a -> a -> a
add x y = x + y
```

With the constraint added to the type, it works fine! What if we use a method from another operation?

```
addWeird :: Num a => a -> a -> a
addWeird x y =
  if x > 1
  then x + y
  else x
```

We get another error, but once again GHC helps us out, as long as we resist the pull of tunnel vision and actually look at what it's telling us:

```
• Could not deduce (Ord a) arising from a
    use of ‘>’
  from the context: Num a
    bound by the type signature for:
```

```
            addWeird ::
            forall a. Num a => a -> a -> a
    at <interactive>:18:1-32
  Possible fix:
    add (Ord a) to the context of
      the type signature for:
        addWeird ::
        forall a. Num a => a -> a -> a
• In the expression: x > 1
  In the expression:
    if x > 1 then x + y else x
  In an equation for ‘addWeird’:
      addWeird x y =
      if x > 1 then x + y else x
```

The problem is that having a `Num` constraint on our type `a` isn't enough. `Num` doesn't imply `Ord`. Given that, we have to add another constraint, which is what GHC tells us to do:

```
addWeird :: (Ord a, Num a) => a -> a -> a
addWeird x y =
  if x > 1
  then x + y
  else x
```

This should type check, because our constraints are asking that `a` have instances of `Num` *and* `Ord`.

Concrete types imply all the type classes they provide

We'll be repurposing some examples from earlier in the chapter, modifying them to all have a concrete type in the place of `a`:

```
add :: Int -> Int -> Int
add x y = x + y

addWeird :: Int -> Int -> Int
addWeird x y =
  if x > 1
  then x + y
  else x

check' :: Int -> Int -> Bool
check' a a' = a == a'
```

These will all type check! This is because the `Int` type has the type classes `Num`, `Eq`, and `Ord` all implemented. We don't need to say `Ord Int => Int -> Int -> Int`, because it doesn't add any information. A concrete type either has a type class instance or it doesn't—adding the constraint means nothing. A concrete type always implies the type classes that are provided for it.

There are some caveats to keep in mind here when it comes to using concrete types. One of the nice things about parametricity and type classes is that you are being explicit about what you mean to do *with* your data, which means you are less likely to make a mistake. `Int` is a big datatype with many inhabitants and many type classes and operations defined for it—it would be easy to make a function that does something unintended. Whereas if we were to write a function, even if we have `Int` values in mind for it, that uses a polymorphic type constrained by the type class instances we want, we could ensure we only use the operations we intend. This isn't a panacea, but sometimes it can be worth avoiding concrete types for these (and other) reasons.

6.14 Chapter exercises

Multiple choice

1. The `Eq` class

    a) includes all types in Haskell.

    b) is the same as the `Ord` class.

c) makes equality tests possible.

d) only includes numeric types.

2. The type class `Ord`

   a) allows any two values to be compared.

   b) is a subclass of `Eq`.

   c) is a superclass of `Eq`.

   d) has no instance for `Bool`.

3. Suppose the type class `Ord` has an operator `>`. What is the type of `>`?

   a) `Ord a => a -> a -> Bool`

   b) `Ord a => Int -> Bool`

   c) `Ord a => a -> Char`

   d) `Ord a => Char -> [Char]`

4. In `x = divMod 16 12`

   a) the type of `x` is `Integer`.

   b) the value of `x` is undecidable.

   c) the type of `x` is a tuple.

   d) `x` is equal to `12 / 16`.

5. The type class `Integral` includes

   a) `Int` and `Integer` numbers.

   b) integral, real, and fractional numbers.

   c) Schrodinger's cat.

   d) only positive numbers.

Does it type check?

For this section of exercises, you'll be practicing looking for type and type class errors.

For example, `printIt` will not work, because functions like `x` have no instance of `Show`, the type class that lets you convert things to `String` (usually for printing):

```
x     :: Int -> Int
x blah = blah + 20

printIt :: IO ()
printIt = putStrLn (show x)
```

Here's the type error you get if you try to load the code:

```
• No instance for (Show (Int -> Int))
    arising from a use of ‘show’
    (maybe you haven't applied a function
     to enough arguments?)
• In the first argument of ‘putStrLn’,
    namely ‘(show x)’
  In the expression: putStrLn (show x)
  In an equation for ‘printIt’:
    printIt = putStrLn (show x)
```

It's saying that it can't find an implementation of the type class `Show` for the type `Int -> Int`, which makes sense. Nothing with the function type constructor, `->`, has an instance of `Show` by default in Haskell.[6]

Examine the following code, and decide whether it will type check. Then load it in GHCi, and see if you were correct. If it doesn't type check, try to match the type error against your understanding of why it didn't work. If you can, fix the error and re-run the code.

1. Does the following code type check? If not, why not?

   ```
   data Person = Person Bool

   printPerson :: Person -> IO ()
   printPerson person = putStrLn (show person)
   ```

2. Does the following type check? If not, why not?

[6] For an explanation and justification of why functions in Haskell cannot have a `Show` instance, see the Haskell Wiki page on this topic: `https://wiki.haskell.org/Show_instance_for_functions`

```
data Mood = Blah
          | Woot deriving Show

settleDown x = if x == Woot
                 then Blah
                 else x
```

3. If you were able to get `settleDown` to type check:

   a) What values are acceptable inputs to that function?

   b) What will happen if you try to run `settleDown 9`? Why?

   c) What will happen if you try to run `Blah > Woot`? Why?

4. Does the following type check? If not, why not?

```
type Subject = String
type Verb = String
type Object = String

data Sentence =
  Sentence Subject Verb Object
  deriving (Eq, Show)

s1 = Sentence "dogs" "drool"
s2 = Sentence "Julie" "loves" "dogs"
```

Given a datatype declaration, what can we do?

Given the following datatype definitions:

```
data Rocks =
  Rocks String deriving (Eq, Show)

data Yeah =
  Yeah Bool deriving (Eq, Show)

data Papu =
  Papu Rocks Yeah
  deriving (Eq, Show)
```

Which of the following will type check? For the ones that don't type check, why don't they?

1. ```
   phew = Papu "chases" True
   ```

2. ```
   truth = Papu (Rocks "chomskydoz")
                (Yeah True)
   ```

3. ```
   equalityForall :: Papu -> Papu -> Bool
   equalityForall p p' = p == p'
   ```

4. ```
   comparePapus :: Papu -> Papu -> Bool
   comparePapus p p' = p > p'
   ```

Match the types

We're going to give you two types and their implementations. Then we're going to ask you if you can substitute the second type for the first. You can test this by typing the first declaration and its type into a file and editing in the new one, loading to see if it fails. *Don't* guess—test all your answers!

1. For the following definition:

   a)
   ```
   i :: Num a => a
   i = 1
   ```

   b) Try replacing the type signature with the following:

   ```
   i :: a
   ```

   After you've formulated your own answer, test that answer. Use GHCi to check what type GHC *infers* for the definitions we provide without a type assigned. For this exercise, you'd type in:

   ```
   Prelude> i = 1
   Prelude> :t i
   -- Result intentionally elided
   ```

2. a)
   ```
   f :: Float
   f = 1.0
   ```

   b)
   ```
   f :: Num a => a
   ```

3. a)
```
f :: Float
f = 1.0
```
   b)
```
f :: Fractional a => a
```

4. Hint for the following: type `:info RealFrac` in your REPL:

   a)
```
f :: Float
f = 1.0
```
   b)
```
f :: RealFrac a => a
```

5. a)
```
freud :: a -> a
freud x = x
```
   b)
```
freud :: Ord a => a -> a
```

6. a)
```
freud' :: a -> a
freud' x = x
```
   b)
```
freud' :: Int -> Int
```

7. a)
```
myX = 1 :: Int

sigmund :: Int -> Int
sigmund x = myX
```
   b)
```
sigmund :: a -> a
```

8. a)
```
myX = 1 :: Int

sigmund' :: Int -> Int
sigmund' x = myX
```
   b)
```
sigmund' :: Num a => a -> a
```

9. a) You'll need to import `sort` from `Data.List`:
```
jung :: Ord a => [a] -> a
jung xs = head (sort xs)
```
   b)
```
jung :: [Int] -> Int
```

10. a)
```
young :: [Char] -> Char
young xs = head (sort xs)
```
   b)
```
young :: Ord a => [a] -> a
```

11. a)
```
mySort :: [Char] -> [Char]
mySort = sort

signifier :: [Char] -> Char
signifier xs = head (mySort xs)
```
    b)
```
signifier :: Ord a => [a] -> a
```

Type-Kwon-Do Two: Electric typealoo

Round two! Same rules apply—you're trying to fill in terms (code) which will fit the types. The idea with these exercises is that you'll derive the implementations from the type information. You'll probably need to use stuff from `Prelude`:

1.
```
chk :: Eq b => (a -> b) -> a -> b -> Bool
chk = ???
```

2. Hint: use some arithmetic operation to combine values of type `b`. Pick one:

```
arith :: Num b
      => (a -> b)
      -> Integer
      -> a
      -> b
arith = ???
```

6.15 Definitions

1. *Type class inheritance* is when a type class has a superclass. This is a way of expressing that a type class requires *another* type class to be available for a given type before you can write an instance:

```
class Num a => Fractional a where
  (/) :: a -> a -> a
  recip :: a -> a
  fromRational :: Rational -> a
```

   Here, the type class `Fractional` *inherits* from `Num`. We could also say that `Num` is a *superclass* of `Fractional`. The long and short of it

is that if you want to write an instance of `Fractional` for some `a`, that type `a` must already have an instance of `Num` before you may do so. Even though, in principle, this example could work, it will fail, because `Nada` doesn't have a `Num` instance:

```
newtype Nada =
  Nada Double deriving (Eq, Show)

instance Fractional Nada where
  (Nada x) / (Nada y) = Nada (x / y)
  recip (Nada n) = Nada (recip n)
  fromRational r = Nada (fromRational r)
```

Then, if you try to load it:

```
• No instance for (Num Nada)
    arising from the superclasses of an
      instance declaration
• In the instance declaration for
    ‘Fractional Nada’
```

You need a `Num` instance first. Can't write one that makes sense? Then you're not allowed to have a `Fractional` instance, either. Them's the rules.

2. *Effects* are how we refer to *observable* actions programs may take other than compute a value. If a function modifies some state or interacts with the outside world in a manner that can be observed, then we say it has an *effect* on the world.

3. `IO` is the type for values whose evaluation bears the possibility of causing side effects, such as printing text, reading text input from the user, reading or writing to files, or connecting to remote computers. This will be explained in *much* more depth in the chapter on `IO`.

4. An *instance* is the definition of how a type class should work for a given type. Instances are unique for a given combination of type class and type.

5. In Haskell, we have *derived instances* so that obvious or common type classes, such as `Eq`, `Enum`, `Ord`, and `Show` can have their instances generated based solely on how a datatype is defined. Programmers can make use of these conveniences without writing the code themselves, over and over.

6.16 Type class inheritance, partial

This is not a complete chart of type class inheritance. However, it illustrates the relationships between a few of the type classes we discussed in this chapter. You can see, for example, that the subclass `Fractional` inherits from the superclass `Num` but not vice versa. While many types have instances of `Show` and `Read`, they aren't superclasses, so we've left them out of the chart for clarity:

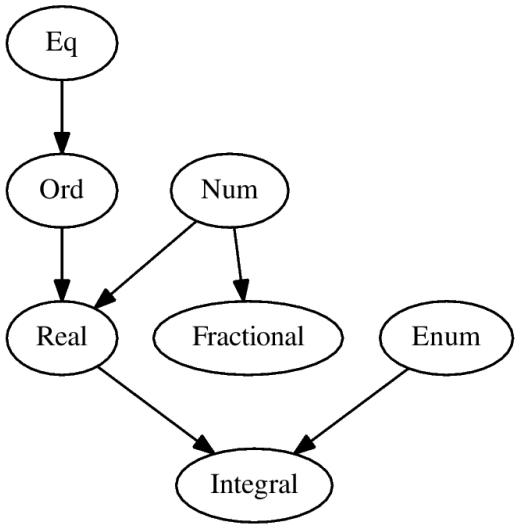


Figure 6.1: Chart of some type classes and their relationships. Only the type classes we've seen so far are included.

6.17 Follow-up resources

1. Philip Wadler and Stephen Blott. *How to make ad-hoc polymorphism less ad hoc*.
2. Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. *Type Classes in Haskell*.
   `http://ropas.snu.ac.kr/lib/dock/HaHaJoWa1996.pdf`

Chapter 7

More Functional Patterns

> I would like to be able to always... divide the things up into as many pieces as I can, each of which I understand separately. I would like to understand the way of adding things up, independently of what it is I'm adding up.
>
> Gerald Sussman

7.1 Make it func-y

You might be asking yourself what this chapter is all about: haven't we been talking about functions all along? We have, but as you might guess from the fact that Haskell is a *functional* programming language, there is more to say—so much more!

A function is an instruction for producing an output from an input, or argument. Functions are *applied* to arguments that bind their parameters to values. The fully applied function with its arguments is then evaluated to produce the output, or result. In this chapter, we will demonstrate that Haskell functions are first-class entities that:

- Can be values in expressions, lists, or tuples.
- Can be passed as arguments to a function.
- Can be returned from a function as a result.
- Make use of syntactic patterns.

7.2 Arguments and parameters

As you know from our discussion of currying, functions in Haskell may appear to have multiple parameters, but this is only the surface appearance. In fact, all functions take one argument and return one result. We construct functions in Haskell through various syntactic means of denoting that an expression takes arguments. Functions are *defined* by the fact that they can be applied to an argument and return a result.

All Haskell values can be arguments to functions. A value that can be used as an argument to a function is a *first-class* value. In Haskell, this includes functions, which can be arguments to more functions still. Not all programming languages allow this, but hopefully the earlier discussion of the function type and currying have given you an idea of how and why this works.

Setting parameters

You name parameters to functions in Haskell by declaring them between the name of the function, which is always at the left margin, and the equals sign, separating the name from both the function

name and the equals sign with white space. The name is a variable, and when we apply the function to an argument, the value of the argument is bound, or unified, with the named parameter in our function definition.

First, we'll define a value with no parameters:

```
myNum :: Integer
myNum = 1

myVal = myNum
```

If we query the type of `myVal`:

```
Prelude> :t myVal
myVal :: Integer
```

The value `myVal` has the same type as `myNum`, because it is equal to it. We can see from the type that it's a value without any parameters, so we can't apply it to anything.

Now, let's introduce a parameter named `f`:

```
myNum :: Integer
myNum = 1

myVal f = myNum
```

And let's see how that changes the type:

```
Prelude> :t myVal
myVal :: t -> Integer
```

By putting `f` after `myVal`, we parameterize `myVal`, which changes the type from `Integer` to `t -> Integer`. The type `t` is polymorphic, because we don't do anything with it—it could be anything. We don't do anything with `f`, so the maximally polymorphic type is inferred. If we do something with `f`, the type will change:

```
Prelude> myNum = 1 :: Integer
Prelude> myVal f = f + myNum
Prelude> :t myVal
myVal :: Integer -> Integer
```

Now it knows that `f` has to be of type `Integer`, because we add it to `myNum`.

We can tell a simple value from a function, in part, because a value is not applied to any arguments, while functions necessarily have parameters that can be applied to arguments.

Although Haskell functions only take one argument per function, we can declare multiple parameters in a term-level function definition:

```
myNum :: Num a => a
myNum = 1
-- [1]

myVal :: Num a => a -> a
myVal f = f + myNum
--       [2]

stillAFunction :: [a] -> [a] -> [a] -> [a]
stillAFunction a b c = a ++ b ++ c
--            [  3  ]
```

1. Declaration of a value with the type `Num a => a`. We can tell it's not a function, because no parameters are named between the name of the declared value and the `=`, so it accepts no arguments, and the value 1 is not a function.

2. Here, `f` is a name for a parameter to the function `myVal`. It represents the possibility of being applied to, or bound to, an input value. The function type is `Num a => a -> a`. If you assign the type `Integer` to `myNum`, as we do above, `myNum` and `myVal` would have the types `Integer` and `Integer -> Integer`, respectively.

3. Here `a`, `b`, and `c` represent parameters for the function. The underlying logic is of nested functions each applied to one argument, rather than one function taking several arguments, but this is how it appears at term level.

Notice what happens to the types as we name more parameters:

```
Prelude> myVal f g = myNum
```

```
Prelude> :t myVal
myVal :: t -> t1 -> Integer

Prelude> myVal f g h = myNum
Prelude> :t myVal
myVal :: t -> t1 -> t2 -> Integer
```

Here, the types are `t`, `t1`, and `t2`, which could be different types. They are allowed but *not required* to be different types. They're all polymorphic, because we give the type inference nothing to go on with respect to which types they could be. The type variables are different, because nothing in our code is preventing them from varying, so they are potentially different types. The inference infers the most polymorphic type that works.

Binding variables to values

Let's consider how the binding of variables works. Applying a function binds its parameters to values. Type parameters become bound to a type, and function variables are bound to a value. The binding of variables concerns not only the application of function arguments, but also things like `let` expressions and `where` clauses. Consider the following function:

```
addOne :: Integer -> Integer
addOne x = x + 1
```

We don't know the result until the `addOne` function is applied to an `Integer` value argument. When `addOne` is applied to a value, we say that `x` is now *bound* to the value the function was applied to. Until a function's arguments have been applied, thereby binding the parameters to values, we cannot make use of the result of the function:

```
addOne 1 -- x is now bound to 1
addOne 1 = 1 + 1
         =     2

addOne 10 -- x is bound to 10
addOne 10 = 10 + 1
          =     11
```

In addition to binding variables through function application, we can use `let` expressions to declare and bind variables as well:

```
bindExp :: Integer -> String
bindExp x =
  let y = 5 in
  "the integer was: " ++ show x
  ++ " and y was: " ++ show y
```

In `show y`, the variable `y` is in scope, because the `let` expression binds `y` to 5. `y` is only in scope *inside* the `let` expression. Let's see something that won't work:

```
bindExp :: Integer -> String
bindExp x =
  let z = y + x in
    let y = 5 in
      "the integer was: "
      ++ show x ++ " and y was: "
      ++ show y ++ " and z was: "
      ++ show z
```

You should see an error, `Variable not in scope: y :: Integer`. We are trying to make `z` equal a value constructed from `x` and `y`. `x` is in scope, because the function argument is visible anywhere in the function. However, `y` is bound in the expression that `let z = …` wraps, so it's not in scope yet—that is, it's not visible to the main function.

In some cases, function arguments are not visible in the function if they have been shadowed. Let's look at a case of *shadowing*:

```
bindExp :: Integer -> String
bindExp x =
  let x = 10; y = 5 in
    "the integer was: " ++ show x
    ++ " and y was: " ++ show y
```

If you apply this to an argument, you'll notice the result never changes:

```
Prelude> bindExp 9001
"the integer was: 10 and y was: 5"
```

This is because the reference to `x` arising from the argument `x` is shadowed by the `x` from the `let` binding. The definition of `x` that is innermost in the code (where the function name at the left margin is the *outside*) takes precedence, because Haskell is *lexically scoped*. Lexical scoping means that resolving the value for a named entity depends on the location in the code and the lexical context, for example in `let` and `where` clauses. Among other things, this makes it easier to know which values are referred to by name and where they come from. Let's annotate the previous example, and we'll see what is meant here:

```
bindExp :: Integer -> String
bindExp x = let x = 10
--     [1]     [2]
               y = 5
            in "x: " ++ show x
--                          [3]
            ++ " y: " ++ show y
```

1. The parameter `x` introduced in the definition of `bindExp`. This gets shadowed by the `x` in `[2]`.
2. This is a `let`-binding of `x` and shadows the definition of `x` introduced as an argument at `[1]`.
3. A use of the `x` bound by `[2]`. Given Haskell's static (lexical) scoping, it will *always* refer to the `x` defined as `x = 10` in the `let` binding!

You can also see the effect of shadowing a name in scope in GHCi using the `let` *statements* you've been kicking around all along:

```
Prelude> x = 5
Prelude> y = x + 5
Prelude> y
10
Prelude> y * 10
100
Prelude> z y = y * 10
Prelude> x
5
Prelude> y
10
Prelude> z 9
90

-- but
Prelude> z y
100
```

Note that while `y` is bound in GHCi's scope to `x + 5`, the introduction of `z y = y * 10` creates a new inner scope that shadows the *name* `y`. Now, when we call `z`, GHCi will use the value we pass as `y` to evaluate the expression, not necessarily the value `10` from the `let` statement `y = x + 5`. Using `y` as an argument to `z`, as in the last example, means the value of `y` from the outer scope is passed to `z` as an argument. The lexically innermost binding for a variable of a particular name always takes precedence. It does not matter that the `y` in `z`'s parameters has the same name as the `y` from earlier in GHCi: `y` will always be bound to the value that `z` is applied to. (Incidentally, the seeming-sequentiality of defining things in GHCi is, under the hood, a never-ending series of nested lambda expressions, similar to the way functions can seem to accept multiple arguments but are, at root, a series of nested functions).

7.3 Anonymous functions

We have already seen how to write *anonymous functions* using the lambda syntax represented by a backslash. Anonymous means "without a name," and that gives us a clue to why we have this syntax—to construct functions and use them without giving them a name.

For example, earlier we looked at this named, i.e., not anonymous, function:

```
triple :: Integer -> Integer
triple x = x * 3
```

And here is the same function but with anonymous function syntax:

```
(\x -> x * 3) :: Integer -> Integer
```

You need the parentheses for the type assertion `:: Integer -> Integer` to apply to the entire anonymous function and not just the `Num a => a` value 3. You can give this function a name, making it not anonymous anymore, in GHCi like this:

```
Prelude> :{
*Main| let trip :: Integer -> Integer
*Main|     trip = \x -> x*3
*Main| :}
Prelude>
```

Similarly, to apply an anonymous function, we'll often need to wrap it in parentheses so that our intent is clear:

```
Prelude> (\x -> x * 3) 5
15
Prelude> \x -> x * 3 1

• Non type-variable argument in the
    constraint: Num (t -> a)
  (Use FlexibleContexts to permit this)
• When checking the inferred type
    it :: forall a t.
      (Num a, Num t, Num (t -> a)) =>
        a -> a
```

You get a type error, because you can't use `Num a => a` values as if they were functions. To the computer, it looks like you're trying to use `3` as a function and apply 3 to 1. Here, the `it` referred to is `3 1`, which it thinks is `3` applied to `1` as if `3` were a function.[1]

Exercises: Grab bag

Note that the following exercises are from source code files, not written for use directly in the REPL. Of course, you can change them to test directly in the REPL, if you prefer.

1. Which (two or more) of the following are equivalent?

   a) `mTh x y z = x * y * z`

   b) `mTh x y = \z -> x * y * z`

   c) `mTh x = \y -> \z -> x * y * z`

   d) `mTh = \x -> \y -> \z -> x * y * z`

2. The type of `mTh` (above) is `Num a => a -> a -> a -> a`. Which is the type of `mTh 3`?

   a) `Integer -> Integer -> Integer`

   b) `Num a => a -> a -> a -> a`

   c) `Num a => a -> a`

   d) `Num a => a -> a -> a`

3. Next, we'll practice writing anonymous lambda syntax.

   For example, one could rewrite:

   ```
   addOne x = x + 1
   ```

   Into:

   ```
   addOne = \x -> x + 1
   ```

   Try to make it so it can still be loaded as a top-level definition by GHCi. This will make it easier to validate your answers.

   a) Rewrite the `f` function in the `where` clause:

[1] In GHCi error messages, `it` refers to the last expression you entered in the REPL.

```
addOneIfOdd n = case odd n of
  True -> f n
  False -> n
  where f n = n + 1
```

b) Rewrite the following to use anonymous lambda syntax:

```
addFive x y = (if x > y then y else x) + 5
```

c) Rewrite the following so that it doesn't use anonymous lambda syntax:

```
mflip f = \x -> \y -> f y x
```

The utility of lambda syntax

You're going to see this anonymous syntax a lot as we proceed through the book, but right now it may not seem to be that useful—it's just another way to write functions.

You may often use this syntax when you're passing a function in as an argument to a higher-order function (more on this soon!), and that's the only place in your program where that particular function will be used. If you're never going to call it, then it doesn't need to be given a name.

7.4 Pattern matching

Pattern matching is an integral and ubiquitous feature of Haskell—so integral and ubiquitous that we've been using it throughout the book without saying anything about it. Once you start, you can't stop.

Pattern matching is a way of matching values against patterns and, where appropriate, binding variables to successful matches. It is worth noting here that *patterns* can include things as diverse as undefined variables, numeric literals, and list syntax. As we will see, pattern matching matches on any and all data constructors.

Pattern matching allows you to expose data and dispatch different behaviors based on that data in your function definitions by deconstructing values to expose their inner workings. There is a reason we describe values as "data *constructors*," although we haven't explored that much yet. Pattern matching also allows us to write functions that can decide between two or more possibilities based on which value it matches.

Patterns are matched against values, or data constructors, *not* types. Matching a pattern may fail, proceeding to the next available pattern to match or succeed. When a match succeeds, the variables exposed in the pattern are bound. Pattern matching proceeds from left to right and outside to inside.

We can pattern match on numbers. In the following example, when the `Integer` argument to the function equals 2, it returns `True`, otherwise `False`:

```
isItTwo :: Integer -> Bool
isItTwo 2 = True
isItTwo _ = False
```

You can enter the same function directly into GHCi using the `:{` and `:}` block syntax—enter `:}` and "return" to end the block:

```
Prelude> :{
*Main| let isItTwo :: Integer -> Bool
*Main|     isItTwo 2 = True
*Main|     isItTwo _ = False
*Main| :}
```

Note the use of the underscore _ after the match against the value 2. This is a means of defining a universal pattern that never fails to match, a sort of "anything else" case:

```
Prelude> isItTwo 2
True
Prelude> isItTwo 3
False
```

Handling all the cases

The order of pattern matches matters! The following version of the function will always return `False`, because it will match the "anything else" case first—and match it to everything—so nothing will get through that to match with the pattern you do want to match:

```
isItTwo :: Integer -> Bool
isItTwo _ = False
isItTwo 2 = True
```

```
    Pattern match is redundant
    In an equation for
      'isItTwo': isItTwo 2 = ...

Prelude> isItTwo 2
False
Prelude> isItTwo 3
False
```

Try to order your patterns from most specific to least specific, particularly as it concerns the use of _ to unconditionally match any value. Unless you get fancy, you should be able to trust GHC's pattern match overlap warning and should triple-check your code when it complains.

What happens if we forget to match a case in our pattern?

```
isItTwo :: Integer -> Bool
isItTwo 2 = True
```

Notice that now our function can only pattern match on the value 2. This is an incomplete pattern match, because it can't match any other data. Incomplete pattern matches applied to data they don't handle will return *bottom*, a non-value used to denote that the program cannot return a value or result. This will throw an exception, which, if unhandled, will make your program fail:

```
Prelude> isItTwo 2
True
Prelude> isItTwo 3
*** Exception: Non-exhaustive patterns in
  function isItTwo
```

We're going to get well acquainted with the idea of bottom in upcoming chapters. For now, it's enough to know that this is what you get when you don't handle all the possible data.

Fortunately, there's a way to know at compile time when your pattern matches are non-exhaustive and don't handle every case:

```
Prelude> :set -Wall
Prelude> :{
```

```
*Main| let isItTwo :: Integer -> Bool
*Main|     isItTwo 2 = True
*Main| :}

    Pattern match(es) are non-exhaustive
    In an equation for ‘isItTwo’:
        Patterns not matched:
          p where p is not one of {2}
```

By turning on all warnings with `-Wall`, we're now told ahead of time that we've made a mistake. Do *not* ignore the warnings GHC provides for you!

Pattern matching against data constructors

Pattern matching serves a couple of purposes. It enables us to vary what our functions do given different inputs. It also allows us to unpack and expose the contents of our data. The values `True` and `False` don't have any other data to expose, but some data constructors have parameters, and pattern matching can let us expose and make use of the data in their arguments.

The next example uses `newtype`, which is a special case of `data` declaration. `newtype` is different in that it permits only one constructor and only one field. We will talk about `newtype` more later. For now, we want to focus on how pattern matching can be used to expose the contents of data and specify behavior based on that data:

```
-- registeredUser1.hs
module RegisteredUser where

newtype Username =
  Username String

newtype AccountNumber =
  AccountNumber Integer

data User =
    UnregisteredUser
  | RegisteredUser Username AccountNumber
```

With the type `User`, we can use pattern matching to accomplish two things. First, `User` is a sum with two constructors, `UnregisteredUser` and `RegisteredUser`. We can use pattern matching to dispatch our function differently depending on which value we get. Then, with the `RegisteredUser` constructor, we see that it is a product of two `newtype` types, `Username` and `AccountNumber`. We can use pattern matching to break down not only the contents of `RegisteredUser` but also that of the `newtype` types, if all the constructors are in scope. Let's write a function to pretty-print `User` values:

```
-- registeredUser2.hs
module RegisteredUser where

newtype Username =
  Username String

newtype AccountNumber =
  AccountNumber Integer

data User =
    UnregisteredUser
  | RegisteredUser Username AccountNumber

printUser :: User -> IO ()
printUser UnregisteredUser =
  putStrLn "UnregisteredUser"

printUser (RegisteredUser
           (Username name)
           (AccountNumber acctNum)) =
  putStrLn $ name ++ " " ++ show acctNum
```

Note that you can continue the pattern on the next line if it gets too long. Next, let's load this into the REPL, and look at the types:

```
Prelude> :l code/registeredUser2.hs
...
Prelude> :t RegisteredUser
RegisteredUser :: Username
```

```
               -> AccountNumber
               -> User
Prelude> :t Username
Username :: String -> Username
Prelude> :t AccountNumber
AccountNumber :: Integer -> AccountNumber
```

Notice how the type of `RegisteredUser` is a function that constructs a `User` out of two arguments: `Username` and `AccountNumber`. This is what we mean when we refer to a value as a "data constructor."

Now, let's use our functions. The argument names are tedious to type in, but they were chosen to ensure clarity. Passing the function an `UnregisteredUser` returns the expected value:

```
Prelude> printUser UnregisteredUser
UnregisteredUser
```

The following, though, asks it to match on data constructor `RegisteredUser` and allows us to construct a `User` out of the `String` value `"callen"` and the `Integer` value `10456`:

```
Prelude> myUser = Username "callen"
Prelude> myAcct = AccountNumber 10456
Prelude> :{
*Main| let rUser =
*Main|       RegisteredUser myUser myAcct
*Main| :}
Prelude> printUser rUser
callen 10456
```

Through the use of pattern matching, we are able to unpack the `RegisteredUser` value of the `User` type and vary behavior over the different constructors of types.

This idea of unpacking and dispatching on data is important, so let us examine another example. First, we're going to write a couple of new datatypes. Writing your own datatypes won't be fully explained until a later chapter, but most of the structure here should be familiar already. We have a sum type called `WherePenguinsLive`:

```
data WherePenguinsLive =
    Galapagos
  | Antarctica
  | Australia
  | SouthAfrica
  | SouthAmerica
  deriving (Eq, Show)
```

And a product type called `Penguin`. We haven't given product types much attention yet, but for now you can think of `Penguin` as a type with only one value, `Peng`, and that value is a sort of box that contains a `WherePenguinsLive` value:

```
data Penguin =
  Peng WherePenguinsLive
  deriving (Eq, Show)
```

Given these datatypes, we will write a couple functions for processing the data:

```
-- is it South Africa? If so, return True
isSouthAfrica :: WherePenguinsLive -> Bool
isSouthAfrica SouthAfrica = True
isSouthAfrica Galapagos = False
isSouthAfrica Antarctica = False
isSouthAfrica Australia = False
isSouthAfrica SouthAmerica = False
```

But that is redundant. We can use _ to indicate an unconditional match on a value we don't care about. The following is better (more concise, easier to read) and does the same thing:

```
isSouthAfrica' :: WherePenguinsLive -> Bool
isSouthAfrica' SouthAfrica = True
isSouthAfrica' _           = False
```

We can also use pattern matching to unpack `Penguin` values to get at the `WherePenguinsLive` value it contains:

```
gimmeWhereTheyLive :: Penguin
                   -> WherePenguinsLive
gimmeWhereTheyLive (Peng whereitlives) =
  whereitlives
```

Try using the `gimmeWhereTheyLive` function on some test data. When you enter the name of the penguin (note the lowercase), it will unpack the `Peng` value to return the `WherePenguinsLive` that's inside:

```
humboldt = Peng SouthAmerica
gentoo = Peng Antarctica
macaroni = Peng Antarctica
little = Peng Australia
galapagos = Peng Galapagos
```

Now, a more elaborate example. We'll expose the contents of `Peng` and match on exactly the `WherePenguinLives` value we care about in one pattern match:

```
galapagosPenguin :: Penguin -> Bool
galapagosPenguin (Peng Galapagos) = True
galapagosPenguin _                = False

antarcticPenguin :: Penguin -> Bool
antarcticPenguin (Peng Antarctica) = True
antarcticPenguin _                 = False
```

In this final function, the `||` operator is an *or* function that will return `True` if either value is `True`:

```
antarcticOrGalapagos :: Penguin -> Bool
antarcticOrGalapagos p =
    (galapagosPenguin p)
  || (antarcticPenguin p)
```

Note that we're using pattern matching to accomplish *two* things here. We're using it to unpack the `Penguin` datatype. We're also specifying which `WherePenguinsLive` value we want to match on.

Pattern matching tuples

You can also use pattern matching, rather than functions, for operating on the contents of tuples. Remember this example from Chapter 4?

```
f :: (a, b) -> (c, d) -> ((b, d), (a, c))
f = undefined
```

When you did that exercise, you may have written it like this:

```
f :: (a, b) -> (c, d) -> ((b, d), (a, c))
f x y = ((snd x, snd y), (fst x, fst y))
```

But we can use pattern matching on tuples to make a somewhat cleaner version of it:

```
f :: (a, b) -> (c, d) -> ((b, d), (a, c))
f (a, b) (c, d) = ((b, d), (a, c))
```

One nice thing about this is that the tuple syntax allows the function to look a great deal like its type. Let's look at more examples of pattern matching on tuples. Note that the *second* example below is *not* a pattern match, but the others are:

```
-- matchingTuples1.hs
module TupleFunctions where
```

These have to be the same type, because `(+)` has the type `a -> a -> a`:

```
addEmUp2 :: Num a => (a, a) -> a
addEmUp2 (x, y) = x + y
```

`addEmUp2` could also be written like so:

```
addEmUp2Alt :: Num a => (a, a) -> a
addEmUp2Alt tup = (fst tup) + (snd tup)

fst3 :: (a, b, c) -> a
fst3 (x, _, _) = x
```

```
third3 :: (a, b, c) -> c
third3 (_, _, x) = x
```

```
Prelude> :l code/matchingTuples1.hs
[1 of 1] Compiling TupleFunctions
Ok, one module loaded.
```

Now, we're going to use GHCi's `:browse` to see a list of the type signatures and functions we load from the module `TupleFunctions`:

```
Prelude> :browse TupleFunctions
addEmUp2 :: Num a => (a, a) -> a
addEmUp2Alt :: Num a => (a, a) -> a
fst3 :: (a, b, c) -> a
third3 :: (a, b, c) -> c

Prelude> addEmUp2 (10, 20)
30
Prelude> addEmUp2Alt (10, 20)
30
Prelude> fst3 ("blah", 2, [])
"blah"
Prelude> third3 ("blah", 2, [])
[]
```

Sweet. Let's do some exercises. Pausing to exercise keeps the muscles flexible, even the mental ones.

Exercises: Variety pack

1. Given the following declarations:

   ```
   k (x, y) = x
   k1 = k ((4-1), 10)
   k2 = k ("three", (1 + 2))
   k3 = k (3, True)
   ```

   a) What is the type of `k`?

   b) What is the type of `k2`? Is it the same type as `k1` or `k3`?

   c) Of `k1, k2, k3`, which will return the number 3 as the result?

2. Fill in the definition of the following function:

```
f :: (a, b, c)
  -> (d, e, f)
  -> ((a, d), (c, f))
f = undefined
```

Remember that tuples have the same syntax for their type constructors and their data constructors.

7.5 Case expressions

Case expressions are a way, similar in some respects to if-then-else expressions, of making a function return a different result based on different inputs. You can use case expressions with any datatype that has visible data constructors. Consider the datatype `Bool`:

```
data Bool = False | True
--   [1]     [2]     [3]
```

1. Type constructor, which we only use in type signatures, not in term-level code like case expressions.
2. Data constructor for the value of `Bool` named `False`—we can match on this.
3. Data constructor for the value of `Bool` named `True`—we can match on this, as well.

Any time we case match or pattern match on a sum type like `Bool`, we should define how we handle each constructor or provide a default that matches all of them. In fact, we *must* handle both cases or use a function that handles both, or we will have written a partial function that can throw an error at runtime. There is *rarely* a good reason to do this: write functions that handle all possible inputs!

Let's start by looking at an if-then-else expression that we saw in a previous chapter:

```
if x + 1 == 1 then "AWESOME" else "wut"
```

We can rewrite this as a case expression, matching on the constructors of `Bool`:

```
funcZ x =
  case x + 1 == 1 of
    True -> "AWESOME"
    False -> "wut"
```

Note that while the syntax is considerably different here, the results will be the same. Be sure to load it in the REPL and try it out.

We could also write a case expression to tell us whether or not something is a palindrome:

```
pal xs =
  case xs == reverse xs of
    True  -> "yes"
    False -> "no"
```

The above can also be written with a `where` clause in cases where you might need to reuse the `y`:

```
pal' xs =
  case y of
    True  -> "yes"
    False -> "no"
  where y = xs == reverse xs
```

In either case, the function will first check if the input string is equal to the reverse of it. If that returns `True`, then the string is a palindrome, so your function says `"yes"`. If not, then it's not.

Here is one more example, also matching on the data constructors from `Bool`, and you can compare its syntax to the if-then-else version we've seen before:

```
-- greetIfCool3.hs
module GreetIfCool3 where

greetIfCool :: String -> IO ()
greetIfCool coolness =
  case cool of
    True  ->
      putStrLn "eyyyyy. What's shakin'?"
    False ->
      putStrLn "pshhhh."
  where cool =
          coolness == "downright frosty yo"
```

So far, the case expressions we've looked at rely on a straightforward pattern match with `True` and `False`, explicitly. In an upcoming section, we'll look at another way to write a case expression.

Exercises: Case practice

We're going to practice using case expressions by rewriting functions. Some of these functions you've seen in previous chapters (and some you'll see later using different syntax, yet again!), but you'll be writing new versions now. Please note, these are all written as they would be in source code files, and we recommend you write your answers in source files, and then load them into GHCi to check, rather than trying to do them directly in the REPL.

First, rewrite if-then-else expressions into case expressions.

1. The following should return `x` when `x` is greater than `y`:

   ```
   functionC x y = if (x > y) then x else y
   ```

2. The following will add 2 to even numbers and otherwise simply return the input value:

   ```
   ifEvenAdd2 n = if even n then (n+2) else n
   ```

   The next exercise doesn't have all the cases covered. See if you can fix it.

3. The following compares a value, `x`, to 0 and returns an indicator for whether `x` is a positive number or negative number. What if `x` is 0? You may need to play with the `compare` function a bit to find what to do:

```
nums x =
  case compare x 0 of
    LT -> -1
    GT -> 1
```

7.6 Higher-order functions

Higher-order functions (HOFs) are functions that accept functions as arguments. Functions are values—why couldn't they be passed around like any other values? This is an important component of functional programming and gives us a way to combine functions efficiently.

Let's examine a standard higher-order function, `flip`:

```
Prelude> :t flip
flip :: (a -> b -> c) -> b -> a -> c

-- using (-) as our (a -> b -> c)
Prelude> (-) 10 1
9
Prelude> fSub = flip (-)
Prelude> fSub 10 1
-9
Prelude> fSub 5 10
5
```

The first parameter of `flip` is a function, such as the `-` operator, that itself has two parameters. `flip` flips the order of the arguments to a function.

We can implement `flip` like this, using the variable `f` to represent the function `(a -> b -> c)`:

```
flip :: (a -> b -> c) -> b -> a -> c
flip f x y = f y x
```

Alternately, it could be written like this:

```
myFlip :: (a -> b -> c) -> b -> a -> c
myFlip f = \ x y -> f y x
```

There's no difference between what `flip` and `myFlip` do: one declares parameters in the function definition, and the other declares them instead in the anonymous function value being returned. But what makes flip a higher-order function? Well, it's this:

```
flip :: (a -> b -> c) -> b -> a -> c
        [     1     ]
flip f x y = f y x
    [2]     [3]
```

1. When we want to express a function argument within a function type, we must use parentheses to nest it.
2. The argument `f` is the function `a -> b -> c`.
3. We apply `f` to `x` and `y`, but `flip` will flip the order of application and apply `f` to `y` and then `x` instead of the usual order.

To better understand how HOFs work syntactically, it's worth remembering how parentheses *associate* in type signatures.

Let's look at the type of the following function:

```
returnLast :: a -> b -> c -> d -> d
returnLast _ _ _ d = d
```

If we explicitly parenthesize `returnLast`, it must match the associativity of `->`, which is right-associative. The following parenthesization works fine. Note that this makes the default currying explicit:

```
returnLast' :: a -> (b -> (c -> (d -> d)))
returnLast' _ _ _ d = d
```

However, this will not work. This is not how `->` associates:

```
returnBroke :: (((a -> b) -> c) -> d) -> d
returnBroke _ _ _ d = d
```

If you attempt to load `returnBroke`, you'll get a type error:

```
• Couldn't match expected type ‘d’
              with actual type
                ‘p0 -> p1 -> p2 -> p2’
  ‘d’ is a rigid type variable bound by
    the type signature for:
      returnBroke ::
        forall a b c d.
          (((a -> b) -> c) -> d) -> d

• The equation(s) for ‘returnBroke’ have
    four arguments,
  but its type
    ‘(((a -> b) -> c) -> d) -> d’ has only
      one
• Relevant bindings include
    returnBroke ::
      (((a -> b) -> c) -> d) -> d
```

This type error is telling us that the type of `returnBroke` only specifies one argument that has the type `((a -> b) -> c) -> d`, yet our function definition seems to expect *four* arguments. The type signature of `returnBroke` specifies a single *function* as the sole argument to `returnBroke`.[2]

We *can* have a type that is parenthesized in that fashion as long as we want to do something different from what `returnLast` does:

```
returnAfterApply :: (a -> b) -> a -> c -> b
returnAfterApply f a c = f a
```

What we're doing here is parenthesizing to the *left* so that we can refer to a separate function, with its own parameters and result, as an argument to our top level function. Here the `(a -> b)` is the `f` argument we use to produce a value of type `b` from a value of type `a`.

One reason we want HOFs is to manipulate how functions are applied to arguments. To understand another reason, let's revisit the `compare` function from the `Ord` type class:

[2] Fun fact: `returnBroke` is an impossible function.

```
Prelude> :t compare
compare :: Ord a => a -> a -> Ordering
Prelude> :info Ordering
data Ordering = LT | EQ | GT
Prelude> compare 10 9
GT
Prelude> compare 9 9
EQ
Prelude> compare 9 10
LT
```

Now, we'll write a function that makes use of this:

```
data Employee = Coder
              | Manager
              | Veep
              | CEO
              deriving (Eq, Ord, Show)

reportBoss :: Employee -> Employee -> IO ()
reportBoss e e' =
  putStrLn $ show e ++
             " is the boss of " ++
             show e'

employeeRank :: Employee
             -> Employee
             -> IO ()

employeeRank e e' =
  case compare e e' of
    GT -> reportBoss e e'
--  [         1         ]
    EQ -> putStrLn "Neither employee\
                   \ is the boss"
--  [               2               ]
    LT -> (flip reportBoss) e e'
--  [            3            ]
```

The `case` in the `employeeRank` function is a case expression. This function says:

1. In the case of comparing `e` and `e'` and finding `e` is greater than `e'`, return `reportBoss e e'`.
2. In the case of finding them equal, return the string `"Neither employee is the boss."`
3. In the case of finding `e` less than `e'`, flip the function `reportBoss`. This could also have been written `reportBoss e' e`.

The `compare` function uses the behavior of the `Ord` instance defined for a given type in order to compare them. In this case, our data declaration lists them in order from `Coder` in the lowest rank and `CEO` in the top rank, so `compare` will use that ordering to evaluate the result of the function.

If we load this up and try it out:

```
Prelude> employeeRank Veep CEO
CEO is the boss of Veep
```

That's probably true in most companies! Being industrious programmers, we naturally want to refactor this a bit to be more flexible—notice how we change the type of `employeeRank`:

```
data Employee = Coder
              | Manager
              | Veep
              | CEO
              deriving (Eq, Ord, Show)

reportBoss :: Employee -> Employee -> IO ()
reportBoss e e' =
  putStrLn $ show e ++
             " is the boss of " ++
             show e'
```

```
employeeRank :: ( Employee
                -> Employee
                -> Ordering )
              -> Employee
              -> Employee
              -> IO ()

employeeRank f e e' =
  case f e e' of
    GT -> reportBoss e e'
    EQ -> putStrLn "Neither employee\
                   \ is the boss"
    LT -> (flip reportBoss) e e'
```

Now, our `employeeRank` function will accept a function argument with the type `Employee -> Employee -> Ordering`, which we name `f`, in the place where we had `compare` before. You'll notice we have the same case expressions here again. We can get the same behavior we had last time by passing it `compare` as the function argument:

```
Prelude> employeeRank compare Veep CEO
CEO is the boss of Veep
Prelude> employeeRank compare CEO Veep
CEO is the boss of Veep
```

But since we're clever hackers, we can subvert the hierarchy with a comparison function that does something a bit different with the following code:

```
data Employee = Coder
              | Manager
              | Veep
              | CEO
              deriving (Eq, Ord, Show)

reportBoss :: Employee -> Employee -> IO ()
reportBoss e e' =
putStrLn $ show e ++
           " is the boss of " ++
           show e'
```

```
codersRuleCEOsDrool :: Employee
                    -> Employee
                    -> Ordering

codersRuleCEOsDrool Coder Coder = EQ
codersRuleCEOsDrool Coder _ = GT
codersRuleCEOsDrool _ Coder = LT
codersRuleCEOsDrool e e' =
  compare e e'

employeeRank :: ( Employee
                -> Employee
                -> Ordering )
              -> Employee
              -> Employee
              -> IO ()

employeeRank f e e' =

  case f e e' of
    GT -> reportBoss e e'
    EQ -> putStrLn "Neither employee\
                   \ is the boss"
    LT -> (flip reportBoss) e e'
```

Here, we've created a new function that changes the behavior of the normal `compare` function by pattern matching on our data constructor, `Coder`. In a case where `Coder` is the first value (and the second value is anything—note the underscore used as a catch-all), the result will be `GT`, or greater than. In a case where `Coder` is the second value passed, this function will return `LT`, or *less than*. In any case where `Coder` is not one of the values, `compare` will exhibit its normal behavior. The case expression in the `employeeRank` function is otherwise unchanged.

And here's how that works:

```
Prelude> employeeRank compare Coder CEO
CEO is the boss of Coder
Prelude> cs = codersRuleCEOsDrool
```

```
Prelude> employeeRank cs Coder CEO
Coder is the boss of CEO
Prelude> employeeRank cs CEO Coder
Coder is the boss of CEO
```

If we use `compare` as our `f` argument, then the behavior is unchanged. If, on the other hand, we use our new function, `codersRuleCEOsDrool` as the `f` argument, then the behavior changes and we unleash anarchy in the cubicle farm.

We are able to rely on the behavior of `compare` but make changes in the part we want to change. This is the value of HOFs. They give us the beginnings of a powerful method for reusing and composing code.

Exercises: Artful dodgy

Given the following definitions, tell us what value results from further applications. When you've written down at least some of the answers and think you know what's what, type the definitions into a file, and load them in GHCi to test your answers:

```
-- Types not provided,
-- try filling them in yourself.

dodgy x y = x + y * 10
oneIsOne = dodgy 1
oneIsTwo = (flip dodgy) 2
```

1. For example, given the expression `dodgy 1 0`, what do you think will happen if we evaluate it? If you put the definitions in a file and load them in GHCi, you can do the following to see the result:

   ```
   Prelude> dodgy 1 0
   1
   ```

   Now, attempt to determine what the following expressions reduce to. Do each one in your head, and verify your answer in the REPL after you think you have one:

2. `dodgy 1 1`
3. `dodgy 2 2`
4. `dodgy 1 2`
5. `dodgy 2 1`
6. `oneIsOne 1`
7. `oneIsOne 2`
8. `oneIsTwo 1`
9. `oneIsTwo 2`
10. `oneIsOne 3`
11. `oneIsTwo 3`

7.7 Guards

We have played around with Booleans and expressions that evaluate to their truth value, including if-then-else expressions, which rely on Boolean evaluation to decide between two outcomes. In this section, we will look at another syntactic pattern called *guards* that relies on truth values to decide between two or more possible results.

if-then-else

Let's begin with a quick review of what we learned about if-then-else expressions in Chapter 4. Note, an if-then-else expression is *not* a guard! This is review before we move on to guards themselves. The pattern is this:

```
if <condition>
  then <result if True>
  else <result if False>
```

Where the `if` condition is an expression that results in a `Bool` value. We saw how this allows us to write functions like this:

```
Prelude> x = 0
Prelude> a = "AWESOME"
Prelude> w = "wut"
Prelude> if (x + 1 == 1) then a else w
"AWESOME"
```

The next couple of examples will demonstrate how to use the multiline block syntax for an `if` expression:

```
-- alternatively
Prelude> x = 0
Prelude> :{
Prelude| if (x + 1 == 1)
Prelude|    then "AWESOME"
Prelude|    else "wut"
Prelude| :}
"AWESOME"
```

The indentation isn't required:

```
Prelude> x = 0
Prelude> :{
Prelude| if (x + 1 == 1)
Prelude| then "AWESOME"
Prelude| else "wut"
Prelude| :}
"AWESOME"
```

In the exercises at the end of Chapter 4, you were asked to write a function called `myAbs` that returns the absolute value of a real number. You would have implemented that function with an if-then-else expression similar to the following:

```
myAbs :: Integer -> Integer
myAbs x = if x < 0 then (-x) else x
```

We're going to look at another way to write this using guards.

Writing guard blocks

Guard syntax allows us to write compact functions that allow for two or more possible outcomes depending on the truth of the conditions. Let's start by looking at how we would write `myAbs` with a guard block instead of with an if-then-else:

```
myAbs :: Integer -> Integer
myAbs x
  | x < 0     = (-x)
  | otherwise = x
```

Notice that each guard has its own equals sign. We didn't put one after the argument in the first line of the function definition, because each case needs its own expression to return if its branch succeeds. Now, we'll enumerate the components for clarity:

```
  myAbs :: Integer -> Integer
  myAbs x
-- [1]  [2]
    |   x < 0     =     (-x)
--  [3]  [4]      [5]     [6]
    |   otherwise   =     x
--  [7]   [8]       [9]  [10]
```

1. The name of our function, `myAbs` still comes first.
2. There is one parameter named `x`.
3. Here's where it gets different. Rather than an `=` immediately after the introduction of any parameter(s), we're starting a new line and using the *pipe*, `|`, to begin a guard case.
4. This is the expression we're using to test whether this branch should be evaluated or not. The guard case expression between the `|` and `=` must evaluate to `Bool`.
5. The `=` denotes that we're declaring what expression to return should our `x < 0` be `True`.
6. Then, after the `=`, we have the expression `(-x)`, which will be returned if `x < 0`.

7. Another new line and a `|` to begin a new guard case.
8. `otherwise` is another name for `True`, used here as a fallback in case `x < 0` is `False`.
9. Another `=` to begin declaring the expression to return if we hit the `otherwise` case.
10. We kick `x` back out if it isn't less than 0.

Let's see how this evaluates:

```
Prelude> myAbs (-10)
10
Prelude> myAbs 10
10
```

In the first example, when it is passed a negative number as an argument, it looks at the first guard and sees that `(-10)` is indeed less than 0, evaluates that as `True`, and so returns the result of `(-x)`, in this case, `(-(-10))` or 10. In the second example, it looks at the first guard, sees that 10 does not meet that condition, so it is `False`, and goes to the next guard. The `otherwise` is always `True`, so it returns `x`, in this case, 10. Guards always evaluate sequentially, so your guards should be ordered from the case that is most restrictive to the case that is least restrictive.

Let's look next at a function that will have more than two possible outcomes, in this case the results of a test of sodium (Na) levels in the blood. We want a function that looks at the numbers (the numbers represent mEq/L or milliequivalents per liter) and tells us if the blood sodium levels are normal or not:

```
bloodNa :: Integer -> String
bloodNa x
  | x < 135   = "too low"
  | x > 145   = "too high"
  | otherwise = "just right"
```

We can incorporate different types of expressions into the guard block, as long as each guard can be evaluated to a `Bool` value. For example, the following function takes three numbers representing

the lengths of the sides of a triangle and tells you whether it's a right triangle (using the Pythagorean theorem):

```
-- c is the hypotenuse of
-- the triangle.

isRight :: (Num a, Eq a)
        => a -> a -> a -> String
isRight a b c
   | a^2 + b^2 == c^2 = "RIGHT ON"
   | otherwise        = "not right"
```

And the following function will take your dog's age and tell you how old your dog is in human years:

```
dogYrs :: Integer -> Integer
dogYrs x
   | x <= 0    = 0
   | x <= 1    = x * 15
   | x <= 2    = x * 12
   | x <= 4    = x * 8
   | otherwise = x * 6
```

Why the different numbers? Because puppies reach maturity much faster than human babies do, so a year-old puppy isn't equivalent to a six or seven-year-old child (there is more complexity to this conversion than this function uses, because other factors such as the size of the dog play a role, as well. You can certainly experiment with that, if you like).

We can also use `where` declarations within guard blocks. Let's say you want to give a test that has 100 questions, and you need a simple function for translating the number of questions the student gets right into a letter grade:

```
avgGrade :: (Fractional a, Ord a)
         => a -> Char
avgGrade x
   | y >= 0.9  = 'A'
   | y >= 0.8  = 'B'
   | y >= 0.7  = 'C'
   | y >= 0.59 = 'D'
   | y <  0.59 = 'F'
   where y = x / 100
```

No surprises there. Notice the variable `y` is introduced, not as an argument to the named function, but in the guard block, and is defined in the `where` clause. By defining it there, it is in scope for all the guards above it. There are 100 problems on the hypothetical test, so any `x` we give it will be divided by 100 to return the letter grade.

Also notice that we leave out the `otherwise`; we could use it for the final case but choose instead to use `<`. That is fine, because in our guards, we handle all possible values. It is important to note that GHCi cannot always tell you when you haven't accounted for all possible cases, and it can be difficult to reason about it, so it is wise to use `otherwise` in your final guard.

Remember: you can use `:set -Wall` in GHCi to turn on warnings, and then it will tell you if you have non-exhaustive patterns.

Exercises: Guard duty

1. It is probably clear to you why you wouldn't put an `otherwise` in your top-most guard, but try it with `avgGrade` anyway, and see what happens. It'll be clearer if you rewrite it as an `otherwise` match: `| otherwise = 'F'`. What happens now if you pass 90 as an argument? 75? 60?

2. What happens if you take `avgGrade` as it is written and reorder the guards? Does it still type check and work the same way? Try moving `| y >= 0.7 = 'C'` and passing it the argument 90, which should be an `'A'`. Does it return `'A'`?

3. What does the following function return?

```
pal xs
    | xs == reverse xs = True
    | otherwise        = False
```

a) `xs` written backwards when it's `True`.

b) `True` when `xs` is a palindrome.

c) `False` when `xs` is a palindrome.

d) `False` when `xs` is reversed.

4. What types of arguments can `pal` take?
5. What is the type of the function `pal`?
6. What does the following function return?

```
numbers x
    | x < 0   = -1
    | x == 0  = 0
    | x > 0   = 1
```

a) The value of its argument plus or minus 1.

b) The negation of its argument.

c) An indication of whether its argument is a positive or negative number or 0.

d) Binary machine language.

7. What types of arguments can `numbers` take?
8. What is the type of the function `numbers`?

7.8 Function composition

Function composition is a type of higher-order function that allows us to combine functions such that the result of applying one function gets passed to the next function as an argument. It is a concise style, in keeping with the terse functional style for which Haskell is known. At first, it seems complicated and difficult to unpack, but once you get the hang of it, it's fun! Let's begin by looking at the type signature and what it means:

```
(.) :: (b -> c) -> (a -> b) -> a -> c
--       [1]        [2]     [3]  [4]
```

1. The first argument is a function from `b` to `c`, passed as an argument (thus the parentheses).
2. The second argument is a function from `a` to `b`.
3. The third argument is a value of type `a`, the same `a` that `[2]` expects as an argument.
4. The final value is of type `c`, the same `c` that `[1]` returns as a result.

Then with the addition of one set of parentheses:

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
--       [1]        [2]         [3]
```

In English:

1. Given a function `b` to `c`,
2. and a function `a` to `b`,
3. return a function `a` to `c`.

The result of `(a -> b)` is the argument of `(b -> c)` so this is how we get from an `a` argument to a `c` result. We define the result of one function as the argument of another function.

Next, let's start looking at composed functions and how to read and work with them. The basic syntax of function composition looks like this:

```
(f . g) x = f (g x)
```

This composition operator, `.` when infix or `(.)` when prefix, takes two functions here, named `f` and `g`. The `f` function corresponds to the `(b -> c)` in the type signature, while the `g` function corresponds to the `(a -> b)`. The `g` function is applied to the (polymorphic) `x` argument. The result of that application then passes to the `f` function as its argument. The `f` function is in turn applied to that argument and evaluated to reach the final result.

Let's go step by step through this transformation. We can think of `(.)` as a way of pipelining data through multiple functions. The following composed functions will first add the values in the list together and then negate the result:

```
Prelude> xs = [1, 2, 3, 4, 5]
Prelude> negate . sum $ xs
-15

-- which is evaluated like this
negate . sum $ xs

-- note: this code works as well
negate (sum xs)
negate (15)
-15
```

Notice that we do this directly in our REPL, because the composition operator is already in scope in `Prelude`. The sum of the list is 15. That result gets passed to the `negate` function and returns a result of -15.

You may be wondering why we need the `$` operator. Remember way back when we talked about the precedence of various operators, we said that `$` has a lower precedence than an ordinary function call (white space, usually). Ordinary function application has a precedence of 10 (out of 10). The composition operator has a precedence of 9. If we leave white space as our function application, this would be evaluated like this:

```
negate . sum [1, 2, 3, 4, 5]
negate . 15
```

Because function application has a higher precedence than the composition operator, that function application would happen before the two functions compose. We'd be trying to pass a numeric value where our composition operator needs a function. By using the `$`, we signal that the application to the arguments should happen *after* the functions are already composed.

We can also parenthesize it instead of using the `$` operator. In that case, it looks like this:

```
Prelude> (negate . sum) [1, 2, 3, 4, 5]
-15
```

The choice of whether to use parentheses or the dollar sign isn't important; it is a question of style and ease of reading.

The next example uses two functions, `take` and `reverse`, and is applied to an argument that is a list of numbers from 1 to 10. What we expect to happen is that the list will first be reversed (from 10 to 1) and then the first 5 elements of the new list will be returned as the result:

```
Prelude> take 5 . reverse $ [1..10]
[10,9,8,7,6]
```

Given the next bit of code, how could we rewrite it to use function composition instead of parentheses?

```
Prelude> take 5 (enumFrom 3)
[3,4,5,6,7]
```

We know that we will have to eliminate the parentheses, add the composition operator, and then add the `$` operator. It will end up looking like this:

```
Prelude> take 5 . enumFrom $ 3
[3,4,5,6,7]
```

You may also define it this way, which closer to how composition is typically written in source files:

```
Prelude> f x = take 5 . enumFrom $ x
Prelude> f 3
[3,4,5,6,7]
```

You may be wondering why we should bother with this if it simply does the same thing as nesting functions in parentheses. One reason is that it is quite easy to compose more than two functions this way.

The `filter odd` function is new for us, but it simply filters the odd numbers (you can change it to `filter even` if you wish) out of the list that `enumFrom` builds for us. Finally, `take` will return as the result only the number of elements we have specified as the argument of `take`. Feel free to experiment with varying any of the arguments:

```
Prelude> take 5 . filter odd . enumFrom $ 3
[3,5,7,9,11]
```

As you compose more functions, you will see that nesting all the parentheses would become tiresome. This operator allows us to do away with that. It also allows us to write in an even more terse style known as "point-free."

7.9 Point-free style

Point-free refers to a style of composing functions without specifying their arguments. The "point" in "point-free" refers to the arguments, not (as it may seem) to the function composition operator. In some sense, we add "points" (the operator) to be able to drop points (arguments). Quite often, point-free code is tidier on the page and easier to read, as it helps the reader focus on the *functions* rather than the data that is being shuffled around.

We said above that function composition looks like this:

```
(f . g) x = f (g x)
```

As you put more functions together, composition can make them easier to read. For example, `(f. g. h) x` can be easier to read than `f (g (h x))`, and it also brings the focus to the functions rather than the arguments. Point-free is an extension of that idea, but now we drop the argument altogether:

```
f . g = \x -> f (g x)
```

```
f . g . h = \x -> f (g (h x))
```

To see what this looks like in practice, we'll start by rewriting in point-free style some of the functions we used in the section above:

```
Prelude> f = negate . sum
Prelude> f [1, 2, 3, 4, 5]
-15
```

Notice that when we define our function `f`, we don't specify that there will be any arguments. Yet, when we apply the function to an argument, the same thing happens as before.

How would we rewrite this:

```
f :: Int -> [Int] -> Int
f z xs = foldr (+) z xs
```

As a point-free function?

```
Prelude> f = foldr (+)
Prelude> f 0 [1..5]
15
```

And now, because we name the function, it can be reused with different arguments.

Here is another example of a short point-free function and its result. It involves a new use of `filter` that uses the `Bool` operator, `==`. Look at it carefully and, on paper or in your head, walk through the evaluation process involved:

```
Prelude> f = length . filter (== 'a')
Prelude> f "abracadabra"
5
```

Next, we'll look at a set of functions that work together, in a single module, and rely on both composition and point-free style:

```
-- arith2.hs
module Arith2 where

add :: Int -> Int -> Int
add x y = x + y

addPF :: Int -> Int -> Int
addPF = (+)

addOne :: Int -> Int
addOne = \x -> x + 1

addOnePF :: Int -> Int
addOnePF = (+1)
```

```
main :: IO ()
main = do
  print (0 :: Int)
  print (add 1 0)
  print (addOne 0)
  print (addOnePF 0)
  print ((addOne   . addOne) 0)

  print ((addOnePF . addOne) 0)
  print ((addOne   . addOnePF) 0)
  print ((addOnePF . addOnePF) 0)
  print (negate (addOne 0))
  print ((negate . addOne) 0)
  print ((addOne . addOne . addOne
          . negate . addOne) 0)
```

Take your time and work through what each function is doing, whether on paper or in your head. Then load this code as a source file, and run it in GHCi to see if your results were accurate.

You should now have a good understanding of how you can use the `(.)` operator to *compose* functions. It's important to remember that the functions in composition are applied from right to left, like a Pacman munching from the right side, reducing the expressions as he goes.

7.10 Demonstrating composition

You may recall back in Chapter 3, we mentioned that the functions `print` and `putStr` seem similar on the surface but behave differently, because they have different underlying types. Let's take a closer look at that now.

First, `putStrLn` and `putStr` have the same type:

```
putStr :: String -> IO ()
putStrLn :: String -> IO ()
```

But the type of `print` is different:

```
print :: Show a => a -> IO ()
```

They all return a result of `IO ()` for reasons we discussed in the previous chapter. But the parameters here are quite different. The first two take `Strings` as arguments, while `print` has a constrained polymorphic parameter, `Show a => a`. The first two work fine if we need to display values that are already of type `String`. But how do we display numbers (or other non-string values)? First, we have to convert those numbers to strings, and then we can print the strings.

You may also recall a function from our discussion of the `Show` type class called `show`. Here's the type of `show` again:

```
show :: Show a => a -> String
```

Fortunately, it was understood that combining `putStrLn` and `show` would be a common pattern, so the function named `print` is the composition of `show` and `putStrLn`. We do it this way, because it's *simpler*. The printing function concerns itself only with printing, while the stringification function concerns itself only with that.

Let's look at two ways to implement `print` with `putStrLn` and `show`. The first uses normal function application:

```
print :: Show a => a -> IO ()
print a = putStrLn (show a)
```

And this one uses `(.)` to compose the constituent functions together:

```
-- type of the . operator
(.) :: (b -> c) -> (a -> b) -> a -> c

print :: Show a => a -> IO ()
print a = (putStrLn . show) a
```

Now, let's go step by step through this use of `(.)`, `putStrLn`, and `show`:

```
(.)       :: (b -> c) -> (a -> b) -> a -> c

putStrLn  :: String -> IO ()
--             [1]      [2]
```

```
show            :: Show a => a -> String
--                      [3]       [4]

putStrLn . show :: Show a => a -> IO ()
--                  [5]           [6]

(.) :: (b -> c) -> (a -> b) -> a -> c
--     [1]  [2]    [3]  [4]   [5]  [6]
```

We can also replace the variables with the specific types they take on in this application of `(.)`:

```
(.) :: Show a => (String -> IO ())
              -> (a -> String)
              ->  a -> IO ()

(.) :: (b      -> c)
    -- (String -> IO ())

    -> (a -> b)
    -- (a -> String)

    ->  a -> c
    --  a -> IO ()
```

1. The string that `putStrLn` accepts as an argument.
2. The `IO ()` that `putStrLn` returns, that is, that performs the side effect of printing and returning unit.
3. A type `a` that must implement the `Show` type class; this is the `Show a => a` from the `show` function, which is a method on the `Show` type class.
4. The string that `show` returns. This is what the `Show a => a` value gets stringified into.
5. The `Show a => a` that the final composed function expects.
6. The `IO ()` that the final composed function returns.

We can now make it point-free. When we are working with functions primarily in terms of *composition* rather than *application*, the point-free version can sometimes (not always) be more elegant.

Here's the previous version of the function:

```
print :: Show a => a -> IO ()
print a = (putStrLn . show) a
```

And here's the point-free version of `print`:

```
print :: Show a => a -> IO ()
print = putStrLn . show
```

The point of `print` is to compose `putStrLn` and `show` so that we don't have to call `show` on its argument ourselves. That is, `print` is principally about the composition of two functions, so it comes out nicely as a point-free function. Saying that we could apply `putStrLn . show` to an argument in this case is redundant.

7.11 Chapter exercises

Multiple choice

1. A polymorphic function:

    a) Changes things into sheep when invoked.

    b) Has multiple arguments.

    c) Has a concrete type.

    d) May resolve to values of different types, depending on inputs.

2. Two functions named `f` and `g` have types `Char -> String` and `String -> [String]`, respectively. The composed function `g . f` has the type:

    a) `Char -> String`

    b) `Char -> [String]`

    c) `[[String]]`

    d) `Char -> String -> [String]`

3. A function `f` has the type `Ord a => a -> a -> Bool`, and we apply it to one numeric value. What is the type now?

   a) `Ord a => a -> Bool`

   b) `Num -> Num -> Bool`

   c) `Ord a => a -> a -> Integer`

   d) `(Ord a, Num a) => a -> Bool`

4. A function with the type `(a -> b) -> c`:

   a) Requires values of three different types.

   b) Is a higher-order function.

   c) Must take a tuple as its first argument.

   d) Has its parameters in alphabetical order.

5. Given the following definition of `f`, what is the type of `f True`?

```
f :: a -> a
f x = x
```

   a) `f True :: Bool`

   b) `f True :: String`

   c) `f True :: Bool -> Bool`

   d) `f True :: a`

Let's write code

1. The following function returns the tens digit of an integral argument:

```
tensDigit :: Integral a => a -> a
tensDigit x = d
   where xLast = x `div` 10
         d     = xLast `mod` 10
```

   a) First, rewrite it using `divMod`.

   b) Does the `divMod` version have the same type as the original version?

c) Next, let's change it so that we're getting the hundreds digit instead. You could start it like this (though that may not be the only possibility):

```
hunsD x = d2
    where d  = undefined
    ...
```

2. Implement the following function of the type `a -> a -> Bool -> a` once using a case expression and once with a guard:

```
foldBool :: a -> a -> Bool -> a
foldBool =
  error
  "Error: Need to implement foldBool!"
```

The result is semantically similar to if-then-else expressions but syntactically quite different. Here is the pattern matching version to get you started:

```
foldBool3 :: a -> a -> Bool -> a
foldBool3 x _ False = x
foldBool3 _ y True  = y
```

3. Fill in the definition. Note that the first argument to our function is *also* a function that can be applied to values. Your second argument is a tuple, which can be used for pattern matching:

```
g :: (a -> b) -> (a, c) -> (b, c)
g = undefined
```

4. For this next exercise, you'll experiment with writing point-free versions of existing code. This involves some new information, so read the following explanation carefully.

Type classes are dispatched by type. `Read` is a type class like `Show`, but it is the dual or "opposite" of `Show`. In general, the `Read` type class isn't something you should plan to use, but this exercise is structured to teach you something about the interaction between type classes and types.

The function `read` in the `Read` type class has the type:

```
read :: Read a => String -> a
```

Notice a pattern?

```
read :: Read a => String -> a
show :: Show a => a -> String
```

Type the following code into a source file. Then load it, and run it in GHCi to make sure you understand why the evaluation results in the answers you see:

```
-- arith4.hs
module Arith4 where

roundTrip :: (Show a, Read a) => a -> a
roundTrip a = read (show a)

main = do
  print (roundTrip 4)
  print (id 4)
```

5. Next, write a point-free version of `roundTrip`. (n.b., this refers to the function definition, not to its application in `main`.)

6. We will continue to use the code in `module Arith4` for this exercise, as well.

   When we apply `show` to a value such as `(1 :: Int)`, the `a` that implements `Show` is type `Int`, so GHC will use the `Int` instance of the `Show` type class to stringify our `Int` value 1.

   However, `read` expects a `String` argument in order to return an `a`. The `String` argument that is the first argument to `read` tells the function nothing about what type the de-stringified result should be. In the type signature `roundTrip` currently has, it knows, because the type variables are the same, so the type that is the input to `show` has to be the same type as the output of `read`.

   Your task now is to change the type of `roundTrip` in `Arith4` to `(Show a, Read b) => a -> b`. How might we tell GHC which instance of `Read` to dispatch against the `String`? Make the expression `print (roundTrip 4)` work. You will only need the *has the type* syntax of `::` and parentheses for scoping.

7.12 Definitions

1. *Binding* or *bound* is a common word used to indicate connection, linkage, or association between two objects. In Haskell, we use it to talk about what value a variable has, e.g., a parameter variable is *bound* to an argument value, meaning the value is passed into the parameter as input and each occurrence of that named parameter will have the same value. Bindings as a plurality will usually refer to a collection of variables and functions that can be referenced by name:

```
blah :: Int
blah = 10
```

   Above, the variable `blah` is bound to the value 10.

2. An *anonymous function* is a function that is not bound to an identifier and is instead passed as an argument to another function and/or used to construct another function. See the following examples:

```
\x -> x
-- anonymous version of id


id x = x
-- not anonymous, it's bound to 'id'
```

3. *Currying* is the process of transforming a function that takes multiple arguments into a series of functions that each take one argument and return one result. This is accomplished through nesting. In Haskell, all functions are curried by default. You don't need to do anything special yourself. `curry` and `uncurry` already exist in `Prelude`:

```
curry' :: ((a, b) -> c) -> a -> b -> c
curry' f a b = f (a, b)

uncurry' :: (a -> b -> c) -> ((a, b) -> c)
uncurry' f (a, b) = f a b
```

   An uncurried function takes a tuple of its arguments:

```
add :: (Int, Int) -> Int
add (x, y) = x + y

add' :: Int -> Int -> Int
add' = curry' add
```

A function that appears to take two arguments is actually two functions that each take one argument and return one result. What makes this work is that a function can return another function:

```
f a b = a + b
```

Is equivalent to:

```
f = \a -> (\b -> a + b)
```

4. *Pattern matching* is a syntactic way of deconstructing product and sum types to get at their inhabitants. With respect to products, pattern matching gives you the means for destructuring and exposing the contents of products, binding one or more values contained therein to names. With sums, pattern matching lets you discriminate which inhabitant of a sum you mean to handle in that match. It's best to explain pattern matching in terms of how datatypes work, so we're going to use terminology that you may not fully understand yet. We'll cover this more deeply soon.

   For example, the following is a "nullary" data constructor. It's neither a sum or a product—it's just a single value:

```
data Blah = Blah
```

   Pattern matching on `Blah` can only do one thing:

```
blahFunc :: Blah -> Bool
blahFunc Blah = True

data Identity a =
  Identity a
  deriving (Eq, Show)
```

`Identity` is a unary data constructor. It's still not a product, as it only contains one value. When you pattern match on `Identity`, you can unpack and expose the value:

```
unpackIdentity :: Identity a -> a
unpackIdentity (Identity x) = x
```

Or you can choose to ignore the contents of `Identity`:

```
ignoreIdentity :: Identity a -> Bool
ignoreIdentity (Identity _) = True
```

You can also ignore the whole `Identity` value completely, since you're matching on a unary data constructor and not a sum:

```
ignoreIdentity' :: Identity a -> Bool
ignoreIdentity' _ = True
```

```
data Product a b =
  Product a b
  deriving (Eq, Show)
```

Now we can choose to use none, one, or both of the values in the product of `a` and `b`:

```
productUnpackOnlyA :: Product a b -> a
productUnpackOnlyA (Product x _) = x


productUnpackOnlyB :: Product a b -> b
productUnpackOnlyB (Product _ y) = y
```

Or we can bind them both to a different name:

```
productUnpack :: Product a b -> (a, b)
productUnpack (Product x y) = (x, y)
```

What happens if you try to bind the values in the product to the same name?

```
data SumOfThree a b c =
    FirstPossible  a
  | SecondPossible b
  | ThirdPossible  c
  deriving (Eq, Show)
```

We can discriminate by the inhabitants of the sum and choose to do different things based on which constructor in the sum they are:

```
sumToInt :: SumOfThree a b c -> Integer
sumToInt (FirstPossible _)  = 0
sumToInt (SecondPossible _) = 1
sumToInt (ThirdPossible _)  = 2
```

We can selectively ignore inhabitants of the sum:

```
sumToInt :: SumOfThree a b c -> Integer
sumToInt (FirstPossible _)  = 0
sumToInt _                  = 1
```

We still need to handle every possible value. Pattern matching is about your *data*.

5. *Bottom* is a non-value used to denote that the program cannot return a value or result. The most elemental manifestation of this is a program that loops infinitely. Other forms can involve things like writing a function that doesn't handle all of its inputs and fails on a pattern match. The following are examples of bottom:

```
-- If you apply this to any values,
-- it'll recurse indefinitely
f x = f x


-- It'll a'splode if you pass a False value
dontDoThis :: Bool -> Int
dontDoThis True = 1
```

```
-- morally equivalent to

definitelyDontDoThis :: Bool -> Int
definitelyDontDoThis True = 1
definitelyDontDoThis False = error "oops"

-- but don't use error
-- We'll show you a better way soon
```

Bottom can be useful as a canary for signaling when code paths are being evaluated. We usually do this to determine how lazy a program is or isn't. You'll see a *lot* of this in our chapter on non-strictness later on.

6. *Higher-order functions* are functions which themselves take functions as arguments or return functions as results. Due to currying, technically any function that appears to take more than one argument is higher-order in Haskell.

   This function is technically higher-order, because of currying:

   ```
   Int -> Int -> Int
   ```

   See? It returns another function after applying the first argument:

   ```
   Int -> (Int -> Int)
   ```

   The rest of the following examples are types of higher-order functions:

   ```
   (a -> b) -> a -> b


   (a -> b) -> [a] -> [b]


   (Int -> Bool) -> [Int] -> [Bool]
   ```

   This function is also higher-order, since it takes a function argument that is higher-order:

   ```
   ((a -> b) -> c) -> [a] -> [c]
   ```

7. *Composition* is the application of a function to the result of having applied another function. The composition operator is a higher-order function, as it takes the functions it composes as arguments and then returns a function of the composition:

```
(.) :: (b -> c) -> (a -> b) -> a -> c

-- is

(.) :: (b -> c) -> (a -> b) -> (a -> c)

-- or

(.) :: (b -> c) -> ((a -> b) -> (a -> c))

-- can be implemented as
comp :: (b -> c) -> ((a -> b) -> (a -> c))
comp f g x = f (g x)
```

   The function `g` is applied to `x`, and the function `f` is applied to the result of `g x`.

8. *Point-free* is programming tacitly, or without mentioning arguments by name. This tends to look like "plumby" code where you're routing data around implicitly or leaving off unnecessary arguments, thanks to currying. The "point" referred to in the term point-free is an argument:

```
-- not point-free
blah x = x
addAndDrop x y = x + 1
reverseMkTuple a b = (b, a)
reverseTuple (a, b) = (b, a)

-- point-free versions of the above
blah = id
addAndDrop = const . (1 +)
reverseMkTuple = flip (,)
reverseTuple = uncurry (flip (,))
```

To see more examples like this, check out the Haskell Wiki page at `https://wiki.haskell.org/Pointfree`.

7.13 Follow-up resources

1. Paul Hudak, John Peterson, and Joseph Fasel. *A Gentle Introduction to Haskell*. Section 4. "Case Expressions and Pattern Matching."
   `https://www.haskell.org/tutorial/patterns.html`
2. Simon Peyton Jones. *The Implementation of Functional Programming Languages*. See p. 53–103.
   `http://research.microsoft.com/en-us/um/people/simonpj/papers/slpj-book-1987/index.htm`
3. Christopher Strachey. *Fundamental Concepts in Programming Languages*. See p. 11 for an explanation of currying.
   `http://www.cs.cmu.edu/~crary/819-f09/Strachey67.pdf`
4. J.N. Oliveira. *An introduction to point-free programming*.
   `http://www.di.uminho.pt/~jno/ps/iscalc_1.ps.gz`
5. Manuel Alcino Pereira da Cunha. *Point-free Program Calculation*.
   `http://www4.di.uminho.pt/~mac/Publications/phd.pdf`

Chapter 8

Recursion

> Imagine a portion of the territory of England has been perfectly levelled, and a cartographer traces a map of England. The work is perfect. There is no particular of the territory of England, small as it can be, that has not been recorded in the map. Everything has its own correspondence. The map, then, must contain a map of the map, that must contain a map of the map of the map, and so on to infinity.
>
> Jorge Luis Borges, citing Josiah Royce

8.1 Recursion

Recursion is defining a function in terms of itself via self-referential expressions. It means that the function will continue to call itself and repeat its behavior until some condition is met to return a result. It's an important concept in Haskell, and in mathematics, because it gives us a means of expressing *indefinite* or incremental computation, without forcing us to explicitly repeat ourselves, and allows the data we are processing to decide when we are done computing.

Recursion is a natural property of many logical and mathematical systems, including human language. That there is no limit on the number of expressible, valid sentences in human language is due to recursion. A sentence in English can have another sentence nested within it. Sentences can be roughly described as structures that have a noun phrase, a verb phrase, and optionally another sentence. This possibility for unlimited, nested sentences is recursive and enables the limitless expressibility therein. Recursion is a means of expressing code that must take an *indefinite* number of steps to return a result.

But the lambda calculus does not appear on the surface to have any means of recursion, because of the anonymity of expressions. How do you call something without a name? Being able to write recursive functions, though, is essential to Turing completeness. We use a combinator—known as the Y combinator or fixed-point combinator—to write recursive functions in the lambda calculus. Haskell has native recursion based on the same principle as the Y combinator.

It is important to have a solid understanding of the behavior of recursive functions. In later chapters, we will see that, in fact, it is not often necessary to write our own recursive functions, as many standard higher-order functions have built-in recursion. But without understanding the systematic behavior of recursion itself, it can be difficult to reason about those HOFs. In this chapter, we will:

- Explore what recursion is and how recursive functions evaluate.
- Go step-by-step through the process of writing our own recursive functions.
- Have fun with *bottom*.

8.2 Factorial!

One of the classic introductory functions for demonstrating recursion in functional languages is factorial. In arithmetic, you might have seen expressions like `4!`. The *bang* next to the number 4 is the notation for the factorial function.

Let's evaluate `4!`:

```
4! = 4 * 3 * 2 * 1
       12 * 2 * 1
           24 * 1
               24
4! = 24
```

Next, let's do it the silly way in Haskell:

```
fourFactorial :: Integer
fourFactorial = 4 * 3 * 2 * 1
```

This will return the correct result, but it only covers one possible result for `factorial`. This is less than ideal. We want to express the general idea of the function, not encode specific inputs and outputs manually.

Let's look at some broken code to introduce the concept of a *base case*:

```
-- This won't work. It never stops.
brokenFact1 :: Integer -> Integer
brokenFact1 n = n * brokenFact1 (n - 1)
```

Let's apply this to 4 and see what happens:

```
brokenFact1 4 =
  4 * (4 - 1)
  * ((4 - 1) - 1)
  * (((4 - 1) - 1) - 1)
  -- this series never stops
```

The way we can stop a recursive expression is by having a base case that stops the self-application to further arguments. Understanding this is critical for writing functions that are correct and terminate properly. Here's what this looks like for `factorial`:

```
module Factorial where

factorial :: Integer -> Integer
factorial 0 = 1
factorial n = n * factorial (n - 1)

brokenFact1 4 =
  4 * (4 - 1)
  * ((4 - 1) - 1)
  * (((4 - 1) - 1) - 1)
  * ((((4 - 1) - 1) - 1) - 1)
  * (((((4 - 1) - 1) - 1) - 1) - 1)
  -- never stops
```

But the base case `factorial 0 = 1` in the fixed version gives our function a stopping point, so the reduction changes:

```
-- Changes to
-- n = n * factorial (n - 1)
factorial 4 =
  4 * factorial (4 - 1)

  -- evaluate (-) applied to 4 and 1
  4 * factorial 3

  -- evaluate factorial applied to 3
  -- expands to 3 * factorial (3 - 1)
  4 * 3 * factorial (3 - 1)

  -- beta reduce (-) applied to 3 and 1
  4 * 3 * factorial 2

  -- evaluate factorial applied to 2
  4 * 3 * 2 * factorial (2 - 1)

  -- evaluate (-) applied to 2 and 1
  4 * 3 * 2 * factorial 1

  -- evaluate factorial applied to 1
  4 * 3 * 2 * 1 * factorial (1 - 1)
```

```
-- evaluate (-) applied to 1 and 1
-- we know factorial 0 = 1
-- so we evaluate that to 1
4 * 3 * 2 * 1 * 1

-- And when we evaluate
-- our multiplications
24
```

Making our base case an identity value for the function (multiplication in this case) means that applying the function to that case doesn't change the result of previous applications.

Another way to look at recursion

In the last chapter, we looked at the compose operator, `(.)`, which is a higher-order function. Function composition is a way of tying two (or more) functions together such that the result of applying the first function gets passed as an argument to the next function. This is the same thing recursive functions are doing—taking the result of the first application of a function and passing it to the next function—except in the case of recursive functions, the first result gets passed back to the same function rather than a different one, until it reaches the base case and terminates.

Where function composition as we normally think of it is static and definite, recursive compositions are indefinite. The number of times the function may be applied depends on the arguments to the function, and the applications can be infinite if a stopping point is not clearly defined.

Let's recall that function composition has the following type:

```
(.) :: (b -> c) -> (a -> b) -> a -> c
```

And when we use it like this:

```
take 5 . filter odd . enumFrom $ 3
```

We know that the first result will be a list generated by `enumFrom`, which will be passed to `filter odd`, giving us a list of only the odd results, and that list will be passed to `take 5`, and our final result will

be the first five members of that list. Thus, results get piped through a series of functions.

Recursion is self-referential composition.[1] We apply a function to an argument, then pass that result on as an argument to a second application of the same function, and so on.

Now look again at how `(.)` is defined:

```
(.)    :: (b -> c) -> (a -> b) -> a -> c
(.) f g = \x -> f (g x)
```

A programming language, such as Haskell, that is built purely on the lambda calculus has one verb for expressing computations that can be evaluated: *apply*. We apply a function to an argument. Applying a function to an argument and potentially doing something with the result is all we can do, no matter what syntactic conveniences we construct to make it seem like we are doing more than that. While we give function composition a special name and operator to point up the pattern and make it convenient to use, it's only a way of saying:

- Given two functions, `f` and `g`, as arguments to `(.)`,
- when we get an argument `x`, apply `g` to `x`,
- then apply `f` to the result of `(g x)`,
- or, to rephrase in code:

  ```
  (.) f g = \x -> f (g x)
  ```

With function recursion, you might notice that it is a form of function application in the same way that composition is a form of function application. The difference is that instead of a fixed number of applications, recursive functions rely on inputs to determine when to stop applying functions to successive results. Without a specified stopping point, the result of `(g x)` will keep being passed back to `g` indefinitely.

Let's look at some code to see the similarity in patterns:

[1] Many thanks to George Makrydakis for discussing this with us.

```
inc :: Num a => a -> a
inc = (+1)

three = inc . inc . inc $ 0
-- different syntax, same thing
three' = (inc . inc . inc) 0
```

Our composition of `inc` bakes the number of applications into the source code. We don't presently have a means of changing how many times we want it to apply `inc` without writing a new function.

So, we might want to make a general function that can apply `inc` an indefinite number of times and allow us to specify as an argument how many times it should be applied:

```
incTimes :: (Eq a, Num a) => a -> a -> a
incTimes 0 n =
  n
incTimes times n =
  1 + (incTimes (times - 1) n)
```

Here, `times` is a variable representing the number of times the incrementing function (not called `inc` here but written as `1 +` in the function body) should be applied to the argument `n`. If we want to apply it 0 times, it will return our `n` back to us. Otherwise, the incrementing function will be applied as many times as we've declared:

```
Prelude> incTimes 10 0
10
Prelude> incTimes 5 0
5
Prelude> incTimes 5 5
10
```

Does this look familiar? In a function such as `incTimes`, the looming threat of unending recursion is minimized, because the number of times to apply the function is an argument to the function itself, and we've defined a stopping point: when `(times - 1)` is equal to 0, it returns `n`, and that's all the applications we get.

We can abstract the recursion out of `incTimes`, too:

```
applyTimes :: (Eq a, Num a) =>
              a -> (b -> b) -> b -> b
applyTimes 0 f b = b
applyTimes n f b = f (applyTimes (n-1) f b)

incTimes' :: (Eq a, Num a) => a -> a -> a
incTimes' times n = applyTimes times (+1) n
```

When we do, we can make the composition more obvious in `applyTimes`:

```
applyTimes :: (Eq a, Num a) =>
              a -> (b -> b) -> b -> b
applyTimes 0 f b =
  b
applyTimes n f b =
  f . applyTimes (n-1) f $ b
```

We're recursively composing our function `f` with `applyTimes (n-1) f` however many subtractions it takes to get `n` to 0!

Intermission: Exercise

Write out the evaluation of the following. It might be a little less noisy if you do so with the form that doesn't use the composition operator, `(.)`:

```
applyTimes 5 (+1) 5
```

8.3 Bottom

⊥, or *bottom*, is a term used in Haskell to refer to computations that do not successfully result in a value. The two main varieties of bottom are computations that fail with an error or those that fail to terminate. In logic, ⊥ corresponds to *false*. Let us examine a few ways in which we can have bottom in our programs:

```
Prelude> let x = x in x
*** Exception: <<loop>>
```

Here, GHCi detects that `let x = x in x` is never going to return a result and short-circuits the never-ending computation.[2] This is an example of bottom, because it is never going to return a result. Note that if you're using a Windows computer, this example may freeze GHCi instead of throwing an exception.

Next, let's define a function that will return an exception:

```
f :: Bool -> Int
f True = error "blah"
f False = 0
```

And let's try that out in GHCi:

```
Prelude> f False
0
Prelude> f True
*** Exception: blah
```

In the first case, when we evaluate `f False` and get 0, that doesn't result in a bottom value. But, when we evaluate `f True`, we get an exception, which is a means of expressing that a computation has failed. We get an exception, because we specify that this value should return an error. But this, too, is an example of bottom.

Another example of a bottom would be a partial function. Let's consider a rewrite of the previous function:

```
f :: Bool -> Int
f False = 0
```

This has the same type and returns the same output. What we've done is elided the `f True = error "blah"` case from the function definition. This is *not* a solution to the problem with the previous function, but it will give us a different exception. We can observe this for ourselves in GHCi:

```
Prelude> f :: Bool -> Int; f False = 0
Prelude> f False
0
```

[2] In GHCi 8.6.5, evaluating this expression may cause your REPL to hang. If you enter it without the `let`, you will get a parse error instead.

```
Prelude> f True
*** Exception: <interactive>:1:19-29:
  Non-exhaustive patterns in function f
```

The `error` value is still there, but our language implementation is making it the fallback case, because we didn't write a *total* function, that is, a function that handles all of its inputs. Because we failed to define ways to handle all potential inputs, for example through an "otherwise" case, the previous function was really:

```
f :: Bool -> Int
f False = 0
f _ = error $ "*** Exception: "
           ++ "Non-exhaustive"
           ++ "patterns in function f"
```

A partial function is one that does not handle all of its inputs. A total function is one that does. How do we make our `f` into a total function? One way is with the use of the datatype `Maybe`:

```
data Maybe a = Nothing | Just a
```

The `Maybe` datatype can take an argument or not. In the first case, `Nothing`, there is no argument. This is our way to say that there is no result or data from the function without hitting bottom. The second case, `Just a`, takes an argument and allows us to return the data we want. `Maybe` makes all uses of `null` values and most uses of bottom unnecessary. Here's how we'd use it with `f`:

```
f :: Bool -> Maybe Int
f False = Just 0
f _ = Nothing
```

Note that the type and both cases all change. Not only do we replace the `error` with the `Nothing` value from `Maybe`, but we also have to wrap 0 in the `Just` constructor from `Maybe`. If we don't do so, we'll get a type error when we try to load the code, as you can see:

```
f :: Bool -> Maybe Int
f False = 0
f _ = Nothing
```

```
Prelude> :l code/brokenMaybe1.hs
[1 of 1] Compiling Main

brokenMaybe1.hs:2:11: error:
• No instance for (Num (Maybe Int)) arising
    from the literal ‘0’
• In the expression: 0
  In an equation for ‘f’: f False = 0
  |
2 | f False = 0
  |           ^
Failed, no modules loaded.
```

This type error is because, as before, 0 has the type `Num a => a`, so it's trying to get an instance of `Num` for `Maybe Int`. We can clarify our intent a bit:

```
f :: Bool -> Maybe Int
f False = 0 :: Int
f _ = Nothing
```

And then get a better type error in the bargain:

```
Prelude> :l code/brokenMaybe2.hs
[1 of 1] Compiling Main

brokenMaybe1.hs:2:11: error:
• Couldn't match expected type ‘Maybe Int’
    with actual type ‘Int’
• In the expression: 0 :: Int
  In an equation for ‘f’:
    f False = 0 :: Int
  |
2 | f False = 0 :: Int
  |           ^^^^^^^^
Failed, no modules loaded.
```

We'll explain `Maybe` in more detail a bit later.

8.4 Fibonacci numbers

Another classic demonstration of recursion in functional programming is a function that, given some value n, calculates the nth number in a Fibonacci sequence. The Fibonacci sequence is a sequence of numbers in which each number is the sum of the previous two: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, and so on. It's an indefinite computation that relies on adding two of its own members, so it's a perfect candidate for a recursive function. We're going to walk through the steps of how we would write such a function for ourselves to get a better understanding of the reasoning process.

Consider the types

The first thing we'll consider is the possible type signature for our function. The Fibonacci sequence only involves positive whole numbers. The argument to our Fibonacci function is going to be a positive whole number, because we're trying to return the nth member of the Fibonacci sequence. Our result will also be a positive whole number, since that's what Fibonacci numbers are. We would be looking, then, for values that are of the `Int` or `Integer` types. We could use one of those concrete types or use a type class for constrained polymorphism. Specifically, we want a type signature that takes one integral argument and returns one integral result. So, our type signature will look something like this:

```
fibonacci :: Integer -> Integer
-- or
fibonacci :: Integral a => a -> a
```

Consider the base case

It may sometimes be difficult to determine your base case up front, but it's worth thinking about. For one thing, you do want to ensure that your function will terminate. For another thing, giving serious consideration to your base case is a valuable part of understanding how your function works. Fibonacci numbers are positive integers, so a reasonable base case is 0. When the recursive process hits zero, it should terminate.

The Fibonacci sequence is a bit trickier than some, though, because it needs two base cases. The sequence has to start off with two numbers, since two numbers are involved in computing the next. The next number after 0 is 1, and we add 0 to 1 to start the sequence so those will be our base cases:

```
fibonacci :: Integral a => a -> a
fibonacci 0 = 0
fibonacci 1 = 1
```

Consider the arguments

We've already determined that the argument to our function, the value to which the function is applied, is an integral number and represents the member of the sequence we want to be evaluated. That is, we want to pass a value such as 10 to this function and have it calculate the 10th number in the Fibonacci sequence. We only need to have one variable as a parameter to this function, then.

But that argument will also be used as an argument within the function due to the recursive process. Every Fibonacci number is the result of adding the preceding two numbers. So, in addition to a variable `x`, we will need to use `(x - 1)` and `(x - 2)` to get both of the numbers before our argument:

```
fibonacci :: Integral a => a -> a
fibonacci 0 = 0
fibonacci 1 = 1
fibonacci x = (x - 1) (x - 2)
-- note: this doesn't work yet.
```

Consider the recursion

All right, now we come to the heart of the matter. In what way will this function refer to itself and call itself? Look at what we've worked out so far: what needs to happen next to produce a Fibonacci number? One thing that needs to happen is that `(x - 1)` and `(x - 2)` need to be added together to produce a result. Try simply adding those two together and running the function that way:

```
fibonacci :: Integral a => a -> a
fibonacci 0 = 0
fibonacci 1 = 1
fibonacci x = (x - 1) + (x - 2)
```

If you pass the value 6 to that function, what will happen?

```
Prelude> fibonacci 6
9
```

Why? Because ((6 - 1) + (6 - 2)) equals 9. But this isn't how we calculate Fibonacci numbers! The sixth member of the Fibonacci sequence is not ((6 - 1) + (6 - 2)). What we want is to add the fifth member of the Fibonacci sequence to the fourth member. That result will be the sixth member of the sequence. We do this by making the function refer to itself. In this case, we have to specify that both `(x - 1)` *and* `(x - 2)` are themselves Fibonacci numbers, so we have to have the function call itself twice:

```
fibonacci  :: Integral a => a -> a
fibonacci 0 = 0
fibonacci 1 = 1
fibonacci x =
  fibonacci (x - 1) + fibonacci (x - 2)
```

Now, if we apply this function to the value 6, we will get a different result:

```
Prelude> fibonacci 6
8
```

Why? Because it evaluates its input recursively:

```
fibonacci 6 = fibonacci 5 + fibonacci 4

fibonacci 5 = fibonacci 4 + fibonacci 3

fibonacci 4 = fibonacci 3 + fibonacci 2

fibonacci 3 = fibonacci 2 + fibonacci 1

fibonacci 2 = fibonacci 1 + fibonacci 0
```

0 and 1 are defined as being equal to 0 and 1. So at this point, our recursion stops, and the function starts adding up the result:

```
fibonacci 0 +             0
fibonacci 1 +             1
fibonacci 2 +   (1 + 0 =) 1
fibonacci 3 +   (1 + 1 =) 2
fibonacci 4 +   (1 + 2 =) 3
fibonacci 5 =   (2 + 3 =) 5
fibonacci 6 =   (3 + 5 =) 8
```

It can be daunting at first to think how you would write a recursive function and what it means for a function to call itself. But as you can see, it's useful when a function makes reference to its own results in a repeated fashion.

8.5 Integral division from scratch

Many people learned multiplication by memorizing multiplication tables, usually up to 10 × 10 or 12 × 12. In fact, one can perform multiplication in terms of addition, repeated over and over. Similarly, one can define integral division in terms of subtraction.

Let's think through our recursive division function one step at a time. First, let's consider the types we would want to use for such a function and see if we can construct a reasonable type signature. When we divide numbers, we have a numerator and a denominator. When we calculate 10 ÷ 2, or in code `10 / 5`, to get the answer 2, 10 is the numerator, 5 is the denominator, and 2 is the quotient. So we have at least three numbers here. So, perhaps a type like `Integer -> Integer -> Integer` would be suitable. You could even add some type synonyms to make it more obvious, if you wish:

```
dividedBy :: Integer -> Integer -> Integer
dividedBy = div
```

Instead of having all the types labeled `Integer` we can instead do:

```
type Numerator = Integer
type Denominator = Integer
type Quotient = Integer

dividedBy :: Numerator
          -> Denominator
          -> Quotient
dividedBy = div
```

The `type` keyword, instead of the more familiar `data` or `newtype`, declares a type synonym, or type alias. Those are all `Integer` types, but we can give them different names to make them easier for human eyes to distinguish in type signatures.

For this example, we didn't write out the recursive implementation of `dividedBy` we had in mind. As it turns out, when we write the function, we will want to change the final type signature a bit, for reasons we'll see in a moment. Sometimes, the use of type synonyms can improve the clarity and purpose of your type signatures, so this is something you'll see, especially in more complex code. For our relatively simple function, it may not be necessary.

Next, let's think through our base case. The way we divide in terms of subtraction is by stopping when our result of having repeatedly subtracted is lower than the divisor. If it divides evenly, it'll stop at 0:

```
Solve 20 divided by 4
--    [1] [2]
-- [1]: Dividend or numerator
-- [2]: Divisor or denominator
-- Result is quotient

20 divided by 4 == 20 - 4, 16
                      - 4, 12
                      - 4, 8
                      - 4, 4
                      - 4, 0
-- 0 is less than 4, so we stop.
-- We subtract 5 times, so 20 / 4 == 5
```

Otherwise, we'll have a remainder. Let's look at a case where it doesn't divide evenly:

```
Solve 25 divided by 4


25 divided by 4 == 25 - 4, 21
                       - 4, 17
                       - 4, 13
                       - 4, 9
                       - 4, 5
                       - 4, 1
We stop at 1, because it's less than 4.
```

In the case of 25 divided by 4, we subtract 4 six times and 1 is our remainder. We can generalize this process of dividing whole numbers, returning the quotient and remainder, into a recursive function that does the repeated subtraction and counting for us. Since we'd like to return the quotient *and* the remainder, we're going to return a 2-tuple—using our friend `(,)`—as the result of our recursive function:

```
dividedBy :: Integral a => a -> a -> (a, a)
dividedBy num denom = go num denom 0
  where go n   d count
        | n < d = (count, n)
        | otherwise =
            go (n - d) d (count + 1)
```

We changed the type signature from the one we had originally worked out, both to make it more polymorphic (`Integral a => a` versus `Integer`) and also to return a tuple instead of just an integer.

Here, we're using a common Haskell idiom called a `go` function. This allows us to define a function via a `where` clause that can accept more arguments than the top-level function `dividedBy` does. In this case, the top-level function takes two arguments, `num` and `denom`, but we need a third argument in order to keep track of how many times we do the subtraction. That argument is called `count` and is defined with a starting value of 0 and is incremented by 1 every time the `otherwise` case is invoked.

There are two branches in our `go` function. The first case is the most specific: when the numerator `n` is less than the denominator `d`, the recursion stops and returns a result. It is not significant that we changed the argument names from `num` and `denom` to `n` and `d`. The `go` function has already been applied to those arguments in the definition of `dividedBy` so the `num`, `denom`, and 0 are bound to `n`, `d`, and `count` in the `where` clause.

The result is a tuple of `count` and the last value for `n`. This is our base case that stops the recursion and gives a final result.

Here's an example of how `dividedBy` expands but with the code inlined:

```
dividedBy 10 2
```

First, we'll do this the previous way, but we'll keep track of how many times we subtract:

```
10 divided by 2 ==
  10 - 2, 8   (subtracted 1 time)
     - 2, 6   (subtracted 2 times)
     - 2, 4   (subtracted 3 times)
     - 2, 2   (subtracted 4 times)
     - 2, 0   (subtracted 5 times)
```

Since the final number is 0, there's no remainder. We subtracted five times. So `10 / 2 == 5`.

Now, we'll expand the code:

```
dividedBy 10 2 =
go 10 2 0

  | 10 < 2 = ...
  -- false, skip this branch

  | otherwise = go (10 - 2) 2 (0 + 1)
```

The `otherwise` above is literally the value `True`, so if the first branch fails, the `otherwise` branch always succeeds:

```
  go 8 2 1
  -- 8 isn't < 2, use the otherwise branch
  go (8 - 2) 2 (1 + 1)
  -- n == 6, d == 2, count == 2

  go 6 2 2
  go (6 - 2) 2 (2 + 1)
  -- 6 isn't < 2, use the otherwise branch
  -- n == 4, d == 2, count == 3

  go 4 2 3
  go (4 - 2) 2 (3 + 1)
  -- 4 isn't < 2, use the otherwise branch
  -- n == 2, d == 2, count == 4

  go 2 2 4
  go (2 - 2) 2 (4 + 1)
  -- 2 isn't < 2, use the otherwise branch
  -- n == 0, d == 2, count == 5

  go 0 2 5
  -- the n < d branch is finally evaluated
  -- because 0 < 2 is true
  -- n == 0, d == 2, count == 5
  | 0 < 2 = (5,  0)

(5, 0)
```

The result of `count` is the quotient, that is, how many times you can subtract 2 from 10. In a case where there is a remainder, that number would be the final value for your numerator and would be returned as the remainder.

8.6 Chapter exercises

Review of types

1. What is the type of `[[True, False], [True, True], [False, True]]`?

a) `Bool`

b) mostly `True`

c) `[a]`

d) `[[Bool]]`

2. Which of the following has the same type as `[[True, False], [True, True], [False, True]]`?

   a) `[(True, False), (True, True), (False, True)]`

   b) `[[3 == 3], [6 > 5], [3 < 4]]`

   c) `[3 == 3, 6 > 5, 3 < 4]`

   d) `["Bool", "more Bool", "Booly Bool!"]`

3. For the function below, which of the following statements are true?

```
func    :: [a] -> [a] -> [a]
func x y = x ++ y
```

   a) `x` and `y` must be of the same type.

   b) `x` and `y` must both be lists.

   c) If `x` is a `String`, then `y` must be a `String`.

   d) All of the above.

4. For the `func` code above, which is a valid application of `func` to both of its arguments?

   a) `func "Hello World"`

   b) `func "Hello" "World"`

   c) `func [1, 2, 3] "a, b, c"`

   d) `func ["Hello", "World"]`

Reviewing currying

Given the following definitions, tell us what value results from further applications:

```
cattyConny :: String -> String -> String
cattyConny x y = x ++ " mrow " ++ y

-- fill in the types

flippy = flip cattyConny

appedCatty = cattyConny "woops"
frappe = flippy "haha"
```

1. What is the value of `appedCatty "woohoo!"`? Try to determine the answer for yourself, then test it in the REPL.
2. `frappe "1"`
3. `frappe (appedCatty "2")`
4. `appedCatty (frappe "blue")`
5. ```
   cattyConny (frappe "pink")
              (cattyConny "green"
                (appedCatty "blue"))
   ```
6. `cattyConny (flippy "Pugs" "are") "awesome"`

Recursion

1. Write out the steps for reducing `dividedBy 15 2` to its final answer according to the Haskell code.
2. Write a function that recursively sums all numbers from 1 to n, n being the argument. So if n is *5*, you'd add 1 + 2 + 3 + 4 + *5* to get 15. The type should be `(Eq a, Num a) => a -> a`.
3. Write a function that multiplies two integral numbers using recursive summation. The type should be `(Integral a) => a -> a -> a`.

Fixing dividedBy

Our `dividedBy` function wasn't quite ideal. For one thing, it is a partial function and doesn't return a result (bottom) when given a divisor that is 0 or less.
   ```

Using the pre-existing `div` function, we can see how negative numbers should be handled:

```
Prelude> div 10 2
5
Prelude> div 10 (-2)
-5
Prelude> div (-10) (-2)
5
Prelude> div (-10) (2)
-5
```

The next issue is how to handle zero. Zero is undefined for division in math, so we ought to use a datatype that lets us say there is no sensible result when the user divides by zero. If you need inspiration, consider using the following datatype to handle this:

```
data DividedResult =
    Result Integer
  | DividedByZero
```

McCarthy 91 function

We're going to describe a function in English, then in math notation, then show you what your function should return for some test inputs. Your task is to write the function in Haskell.

The McCarthy 91 function yields x - 10 when x > 100 and 91 otherwise. The function is recursive:

$$MC(n) = \begin{cases} n - 10 & \text{if } n > 100 \\ MC(MC(n + 11)) & \text{if } n \leq 100 \end{cases}$$

```
mc91 = undefined
```

You haven't seen `map` yet, but all you need to know right now is that it applies a function to each member of a list and returns the resulting list. We'll explain it in more detail in the next chapter:

```
Prelude> map mc91 [95..110]
[91,91,91,91,91,91,91,92,93,94,95,96,97,98,
  99,100]
```

Numbers into words

```
module WordNumber where

import Data.List (intersperse)

digitToWord :: Int -> String
digitToWord n = undefined

digits :: Int -> [Int]
digits n = undefined

wordNumber :: Int -> String
wordNumber n = undefined
```

Here, `undefined` is a placeholder to show you where you need to fill in the functions. The `n` to the right of the function names is the argument that will be an integer.

Fill in the implementations of the functions above so that `wordNumber` returns the English word version of the `Int` value. You will first write a function that turns integers from 0–9 into their corresponding English words: "one," "two," and so on. Then, you will write a function that takes the integer, separates the digits, and returns it as a list of integers. Finally, you will need to apply the first function to the list produced by the second function and turn it into a single string with interspersed hyphens.

We've laid out multiple functions for you to consider as you tackle the problem. You may not need all of them, depending on how you solve it—these are just suggestions. Play with them, and look up their documentation to understand them in greater detail.

You will probably find this difficult:

```
div         :: Integral a => a -> a -> a
mod         :: Integral a => a -> a -> a
map         :: (a -> b) -> [a] -> [b]
concat      :: [[a]] -> [a]
intersperse :: a -> [a] -> [a]
(++)        :: [a] -> [a] -> [a]
(:[])       :: a -> [a]
```

Also consider:

```
Prelude> div 135 10
13
Prelude> mod 135 10
5
Prelude> div 13 10
1
Prelude> mod 13 10
3
```

Here is what your REPL output should look like when it's working:

```
Prelude> wordNumber 12324546
"one-two-three-two-four-five-four-six"
```

8.7 Definitions

1. *Recursion* is a means of computing results that may require an indefinite amount of work to obtain through the use of repeated function application. Most recursive functions that terminate or otherwise do useful work will often have a case that calls itself and a base case that acts as a backstop of sorts for the recursion.

   This function is not recursive:

   ```
   lessOne :: Int -> Int
   lessOne n = n - 1
   ```

   This one is recursive:

   ```
   zero :: Int -> Int
   zero 0 = 0
   zero n = zero (n - 1)
   ```

Chapter 9

Lists

> If the doors of perception were cleansed, everything would appear to man as it is—infinite.
>
> William Blake

9.1 Lists

Lists do double duty in Haskell. The first purpose lists serve is as a way to refer to and process a collection or plurality of values. The second is as an infinite series of values, usually generated by a function, which allows them to act as a stream datatype.

In this chapter, we will:

- Explain the list datatype and how to pattern match on lists.
- Practice many standard library functions for operating on lists.
- Learn about the underlying representation of lists.
- See what that representation means for their evaluation.
- And do a whole bunch of exercises!

9.2 The list datatype

The list datatype in Haskell is defined like this:

```
data [] a = [] |  a : [a]
```

Here, `[]` is the type constructor for lists as well as the data constructor for the empty list. The `[]` data constructor is a nullary constructor, because it takes no arguments. The second data constructor, in contrast, has arguments. `:` is an infix operator usually called "cons" (short for *construct*). The cons operator takes a value of type `a` and a list of type `[a]` and evaluates to `[a]`.

Whereas the list datatype as a whole is a sum type, as we can tell from the `|` in the definition, the second data constructor, `:`, is a *product*, because it takes two arguments. Remember, a sum type can be read as an "or," as in the `Bool` datatype, which can be either `False` or `True`. A product is like an "and." We're going to talk more about sum and product types in another chapter, but for now it will suffice to recognize that `a : [a]` constructs a value from two arguments, by adding the `a` to the front of the list `[a]`. The list datatype is a sum type, though, because it is *either* an empty list *or* a single value with more list—not both.

In English, one can read this as:

```
data [] a = [] | a : [a]
-- [1] [2] [3] [4] [5] [6]
```

1. The datatype with the type constructor `[]`,
2. which takes a single type constructor argument of type `a`,
3. at the term level can be constructed via
4. the nullary list constructor `[]`,
5. *or* it can be constructed by
6. the data constructor (or cons) `:`, which is a product of a value of type `a` from the type constructor *and* a value of type `[a]`, that is, "more list."

The cons constructor, `:`, is an infix data constructor and goes between the two arguments `a` and `[a]` that it accepts. Since it takes two arguments, it is a product of those two arguments, like the tuple type `(a, b)`. Unlike a tuple, however, this constructor is recursive, because it mentions its own type `[a]` as one of the members of the product.

If you're an experienced programmer or took a CS class at some point, you may be familiar with singly-linked lists. This is a fair description of the list datatype in Haskell, although average case performance in some situations changes due to non-strict evaluation; however, it can contain infinite data, which makes it also work as a stream datatype, but one that has the option of ending the stream with the `[]` data constructor.

9.3 Pattern matching on lists

We know we can pattern match on data constructors, and the data constructors for lists are no exceptions. Here, we match on the first argument to the infix `:` constructor, ignoring the rest of the list, and return that value:

```
Prelude> myHead (x : _) = x
Prelude> :t myHead
myHead :: [t] -> t
```

```
Prelude> myHead [1, 2, 3]
1
```

We can do the opposite, as well:

```
Prelude> myTail (_ : xs) = xs
Prelude> :t myTail
myTail :: [t] -> [t]
Prelude> myTail [1, 2, 3]
[2,3]
```

We do need to be careful with functions like these, however. Neither `myHead` nor `myTail` has a case to handle an empty list—if we try to pass them an empty list as an argument, they can't pattern match:

```
Prelude> myHead []
*** Exception:
  Non-exhaustive patterns
    in function myHead

Prelude> myTail []
*** Exception:
  Non-exhaustive patterns
    in function myTail
```

The problem is that the type `[a] -> a` of `myHead` is deceptive, because the `[a]` type doesn't guarantee that it'll have an `a` value. It's not guaranteed that the list will have at least one value, so `myTail` can fail, as well. One possibility is putting in a base case:

```
myTail :: [a] -> [a]
myTail []       = []
myTail (_ : xs) = xs
```

With that addition, our function now evaluates like this:

```
Prelude> myTail [1..5]
[2,3,4,5]
Prelude> myTail []
[]
```

Using Maybe A better way to handle this situation is with a datatype called `Maybe`. We'll save a full treatment of `Maybe` for a later chapter, but this should give you some idea of how it works. The idea here is that it makes your failure case explicit, and as programs get longer and more complex, that can be quite useful.

Let's try an example using `Maybe` with `myTail`. Instead of having a base case that returns an empty list, the function written with `Maybe` would return a result of `Nothing`. As we can see below, the `Maybe` datatype has two potential values, `Nothing` or `Just a`:

```
Prelude> :info Maybe
data Maybe a = Nothing | Just a
```

Rewriting `myTail` to use `Maybe` is fairly straightforward:

```
safeTail        :: [a] -> Maybe [a]
safeTail []     = Nothing
safeTail (_:[]) = Nothing
safeTail (_:xs) = Just xs
```

Notice that our function is still pattern matching on the list. We've made the second base case `safeTail (x:[]) = Nothing` in order to reflect the fact that if your list has only one value inside it, its tail is an empty list. If you leave this case out, then this function will return `Just []` for lists that have only a head value. Take a few minutes to play around with this function and see how it works. Then, see if you can rewrite the `myHead` function above using `Maybe`.

Later in the book, we'll also cover a datatype called `NonEmpty`, which always has at least one value and avoids the empty list problem.

9.4 List's syntactic sugar

Haskell has some syntactic sugar to accommodate the use of lists, so that you can write:

```
Prelude> [1, 2, 3] ++ [4]
[1, 2, 3, 4]
```

Rather than:

```
Prelude> (1 : 2 : 3 : []) ++ 4 : []
[1,2,3,4]
```

The syntactic sugar is here to allow the building of lists in terms of successive applications of the cons operator, `:`, to a value without having to tediously type it all out.

When we talk about lists, we often talk about them in terms of "cons cells" and spines. The syntactic sugar obscures this underlying construction, but looking at the desugared version above may make it more clear. The cons cells are the list datatype's second data constructor, `a : [a]`, the result of recursively prepending a value to "more list." The cons cell is a *conceptual* space that values may inhabit.

The spine is the connective structure that holds the cons cells together and in place. As we will soon see, this structure nests the cons cells rather than ordering them in a right-to-left row. Because different functions may treat the spine and the cons cells differently, it is important to understand this underlying structure.

9.5 Using ranges to construct lists

There are several ways we can construct lists. One of the simplest is with ranges. The basic syntax is to make a list that has the element you want to start the list from followed by two dots followed by the value you want as the final element in the list. Here are some examples using the range syntax, followed by the desugared equivalents using functions from the `Enum` type class:

```
Prelude> [1..10]
[1,2,3,4,5,6,7,8,9,10]
Prelude> enumFromTo 1 10
[1,2,3,4,5,6,7,8,9,10]

Prelude> [1,2..10]
[1,2,3,4,5,6,7,8,9,10]
Prelude> enumFromThenTo 1 2 10
[1,2,3,4,5,6,7,8,9,10]

Prelude> [1,3..10]
[1,3,5,7,9]
```

```
Prelude> enumFromThenTo 1 3 10
[1,3,5,7,9]

Prelude> [2,4..10]
[2,4,6,8,10]
Prelude> enumFromThenTo 2 4 10
[2,4,6,8,10]

Prelude> ['t'..'z']
"tuvwxyz"
Prelude> enumFromTo 't' 'z'
"tuvwxyz"
```

The types of the functions underlying the range syntax are:

```
enumFrom       :: Enum a
               => a -> [a]
enumFromThen   :: Enum a
               => a -> a -> [a]

enumFromTo     :: Enum a
               => a -> a -> [a]
enumFromThenTo :: Enum a
               => a -> a -> a -> [a]
```

All of these functions require that the type being "ranged" have an instance of the `Enum` type class. The first two functions, `enumFrom` and `enumFromThen`, generate lists of indefinite, possibly infinite, length. For it to create an infinitely long list, you must be ranging over a type that has no upper bound in its enumeration. `Integer` is such a type. You can make `Integer` values as large as you have memory to describe them.

Be aware that the first argument to `enumFromTo` must be lower than its second argument, otherwise you'll get an empty list:

```
Prelude> enumFromTo 3 1
[]
Prelude> enumFromTo 1 3
[1,2,3]
```

Exercise: EnumFromTo

Some things you'll want to know about the `Enum` type class:

```
Prelude> :info Enum
class Enum a where
  succ :: a -> a
  pred :: a -> a
  toEnum :: Int -> a
  fromEnum :: a -> Int
  enumFrom :: a -> [a]
  enumFromThen :: a -> a -> [a]
  enumFromTo :: a -> a -> [a]
  enumFromThenTo :: a -> a -> a -> [a]
Prelude> succ 0
1
Prelude> succ 1
2
Prelude> succ 'a'
'b'
```

Write your own `enumFromTo` definitions for the types provided. Do not use range syntax to do so. It should return the same results as if you did `[start..stop]`. Replace the `undefined`, a value that results in an error when evaluated, with your own definition.

```
eftBool :: Bool -> Bool -> [Bool]
eftBool = undefined

eftOrd :: Ordering
       -> Ordering
       -> [Ordering]
eftOrd = undefined

eftInt :: Int -> Int -> [Int]
eftInt = undefined

eftChar :: Char -> Char -> [Char]
eftChar = undefined
```

9.6 Extracting portions of lists

In this section, we'll take a look at some useful functions for extracting portions of a list and dividing lists into parts. The first three functions have similar type signatures, taking `Int` arguments and applying them to a list argument:

```
take :: Int -> [a] -> [a]
drop :: Int -> [a] -> [a]
splitAt :: Int -> [a] -> ([a], [a])
```

We have seen examples of some of the above functions in previous chapters, but they are common and useful enough that they deserve review.

The `take` function takes the specified number of elements out of a list and returns a list containing just those elements. As you can see, it takes one argument that is an `Int` and applies that to a list argument. Here's how it works:

```
Prelude> take 7 ['a'..'z']
"abcdefg"

Prelude> take 3 [1..10]
[1,2,3]

Prelude> take 3 []
[]
```

Notice that when we pass it an empty list as an argument, it returns an empty list. These lists use the syntactic sugar for building lists with ranges. We can also use `take` with a list-building function, such as `enumFrom`. Reminder: `enumFrom` can generate an infinite list if the type of list inhabitants is, such as `Integer`, an infinite set. But as long as we're only taking a certain number of elements from that set, it won't generate an infinite list:

```
Prelude> take 10 (enumFrom 10)
[10,11,12,13,14,15,16,17,18,19]
```

The `drop` function is similar to `take` but drops the specified number of elements off the beginning of the list. Again, we can use it with ranges or list-building functions:

```
Prelude> drop 5 [1..10]
[6,7,8,9,10]

Prelude> drop 8 ['a'..'z']
"ijklmnopqrstuvwxyz"

Prelude> drop 4 []
[]

Prelude> drop 2 (enumFromTo 10 20)
[12,13,14,15,16,17,18,19,20]
```

The `splitAt` function cuts a list into two parts at the element specified by the `Int` and makes a tuple of two lists:

```
Prelude> splitAt 5 [1..10]
([1,2,3,4,5],[6,7,8,9,10])

Prelude> splitAt 10 ['a'..'z']
("abcdefghij","klmnopqrstuvwxyz")

Prelude> splitAt 5 []
([],[])

Prelude> splitAt 3 (enumFromTo 5 15)
([5,6,7],[8,9,10,11,12,13,14,15])
```

The higher-order functions `takeWhile` and `dropWhile` are a bit different, as you can see from the type signatures:

```
takeWhile :: (a -> Bool) -> [a] -> [a]
dropWhile :: (a -> Bool) -> [a] -> [a]
```

So these functions take and drop items out of a list that meet some condition, as we can see from the presence of `Bool`. `takeWhile` will take

elements out of a list that meet that condition and then stop when it meets the first element that doesn't satisfy the condition:

Take the elements that are less than 3:

```
Prelude> takeWhile (<3) [1..10]
[1,2]
```

Take the elements that are less than 8:

```
Prelude> takeWhile (<8) (enumFromTo 5 15)
[5,6,7]
```

The next example returns an empty list, because it stops taking as soon as the condition isn't met, which in this case is the first element:

```
Prelude> takeWhile (>6) [1..10]
[]
```

In the final example below, why does it only return a single `a`?

```
Prelude> takeWhile (=='a') "abracadabra"
"a"
```

Now, we'll look at `dropWhile`. Its behavior is probably predictable based on the functions and type signatures we've already seen in this section. We will use the same arguments as we used with `takeWhile`, so the difference between them is easy to see:

```
Prelude> dropWhile (<3) [1..10]
[3,4,5,6,7,8,9,10]

Prelude> dropWhile (<8) (enumFromTo 5 15)
[8,9,10,11,12,13,14,15]

Prelude> dropWhile (>6) [1..10]
[1,2,3,4,5,6,7,8,9,10]

Prelude> dropWhile (=='a') "abracadabra"
"bracadabra"
```

Exercises: Thy fearful symmetry

1. Using `takeWhile` and `dropWhile`, write a function that takes a string and returns a list of strings, using spaces to separate the elements of the string into words, as in the following sample:

```
Prelude> myWords "sheryl wants fun"
["sheryl", "wants", "fun"]
```

2. Next, write a function that takes a string and returns a list of strings, using newline separators to break up the string, as in the following (your job is to fill in the undefined function):

```
module PoemLines where

firstSen = "Tyger Tyger, burning bright\n"
secondSen = "In the forests of the night\n"
thirdSen = "What immortal hand or eye\n"
fourthSen = "Could frame thy fearful\
          \ symmetry?"
sentences = firstSen ++ secondSen
         ++ thirdSen ++ fourthSen
```

This is the result that `putStrLn sentences` should print:

```
Tyger Tyger, burning bright
In the forests of the night
What immortal hand or eye
Could frame thy fearful symmetry?
```

Implement this:

```
myLines :: String -> [String]
myLines = undefined
```

We want `myLines sentences` to equal:

```
shouldEqual =
  [ "Tyger Tyger, burning bright"
  , "In the forests of the night"
  , "What immortal hand or eye"
  , "Could frame thy fearful symmetry?"
  ]
```

The main function here is a small test to ensure you've written your function correctly:

```
main :: IO ()
main =
  print $
  "Are they equal? "
  ++ show (myLines sentences
           == shouldEqual)
```

3. Now, let's look at what those two functions have in common. Try writing a new function that parameterizes the character you're breaking the string argument on and rewrite `myWords` and `myLines` using that parameter.

9.7 List comprehensions

List comprehensions are a means of generating a new list from a list or lists. They come directly from the concept of set comprehensions in mathematics, including similar syntax. They must have at least one list, called the generator, that gives the input for the comprehension, that is, provides the set of items from which the new list will be constructed. They may have conditions to determine which elements are drawn from the list and/or which functions are applied to those elements.

Let's start by looking at a very simple example:

```
[  x^2   |   x <- [1..10]]
-- [1]  [2]    [   3   ]
```

1. This is the output function that will apply to the members of the list we indicate.

2. The pipe here designates the separation between the output function and the input.
3. This is the input set: a generator list and a variable that represents the elements that will be drawn from that list. This says, "from a list of numbers from 1–10, take (`<-`) each element as an input to the output function."

In plain English, that list comprehension will produce a new list that includes the square of every number from 1 to 10:

```
Prelude> [x^2 | x <- [1..10]]
[1,4,9,16,25,36,49,64,81,100]
```

Now, we'll look at some ways to vary which elements are drawn from the generator list(s).

Adding predicates

List comprehensions can optionally take predicates that limit the elements drawn from the generator list. The predicates must evaluate to `Bool` values, as in other condition-placing function types we've looked at (for example, guards). Then, the items drawn from the list and passed to the output function will only be those that meet the `True` case in the predicate.

For example, let's say we want a similar list comprehension as we used above, but this time we want our new list to contain the squares of only the even numbers while ignoring the odds. In that case, we put a comma after our generator list and add the condition:

```
Prelude> [x^2 | x <- [1..10], rem x 2 == 0]
[4,16,36,64,100]
```

Here, we've specified that the only elements to take from the generator list as `x` are those that, when divided by two, have a remainder of zero—that is, even numbers.

We can also write list comprehensions that have multiple generators. One thing to note is that the rightmost generator will be exhausted first, then the second rightmost, and so on.

For example, let's say you want to make a list of `x` to the `y` power, instead of squaring all of them as we did above. Separate the two inputs with a comma, as below:

```
Prelude> [x^y | x <- [1..5], y <- [2, 3]]
[1,1,4,8,9,27,16,64,25,125]
```

When we examine the resulting list, we see that it is each x value first to the second power and then to the third power, followed by the next x value to the second and then to the third and so on, ending with the result of 5^2 and 5^3. We are applying the function to each possible pairing of values from the two lists we're binding values out of. It begins by trying to get a value out of the leftmost generator, from which we're getting x.

We could put a condition on that, too. Let's say we only want to return the list of values that are less than 200. We add another comma, and write our predicate:

```
Prelude> :{
Prelude| [x ^ y |
Prelude|  x <- [1..10],
Prelude|  y <- [2, 3],
Prelude|  x ^ y < 200]
Prelude| :}
[1,1,4,8,9,27,16,64,25,125,36,49,64,81,100]
```

We can use multiple generators to turn two lists into a list of tuples containing those elements, as well. The generator lists don't even have to be of the same length or, due to the nature of the tuple type, even the same type:

```
Prelude> :{
Prelude| [(x, y) |
Prelude|  x <- [1, 2, 3],
Prelude|  y <- [6, 7]]
Prelude| :}
[(1,6),(1,7),(2,6),(2,7),(3,6),(3,7)]

Prelude> :{
Prelude| [(x, y) |
Prelude| x <- [1, 2, 3],
Prelude| y <- ['a', 'b']]
Prelude| :}
```

```
[(1,'a'),(1,'b'),(2,'a'),
 (2,'b'),(3,'a'),(3,'b')]
```

Again, the pattern is that it generates every possible tuple for the first `x` value, then it moves to the next `x` value, and so on.

Recall that the first list comprehension we looked at generates a list of all the values of x^2 when `x` is a number from 1–10. Let's say you want to use that list in another list comprehension. First, you'd want to give that list a name. Let's call it `mySqr`:

```
Prelude> mySqr = [x^2 | x <- [1..10]]
```

Now we can use that list as the generator for another list comprehension. Here, we will limit our input values to those that are less than four for the sake of brevity:

```
Prelude> mySqr = [x^2 | x <- [1..10]]
Prelude> :{
Prelude| [(x, y) |
Prelude| x <- mySqr,
Prelude| y <- [1..3], x < 4]
Prelude| :}
[(1,1),(1,2),(1,3)]
```

Exercises: Comprehend thy lists

Take a look at the following functions, determine what you think the output lists will be, and then run them in your REPL to verify (note that you will need the `mySqr` list from above in scope to do this):

```
[x | x <- mySqr, rem x 2 == 0]
```

```
[(x, y) | x <- mySqr,
          y <- mySqr,
          x < 50, y > 50]
```

```
take 5 [ (x, y) | x <- mySqr,
                  y <- mySqr,
                  x < 50, y > 50 ]
```

List comprehensions with strings

It's worth remembering that strings are lists, so list comprehensions can also be used with strings. We're going to introduce a standard function called `elem`[1] that tells you whether an element is in a list or not. It evaluates to a `Bool` value, so it is useful as a predicate in list comprehensions:

```
Prelude> :t elem
elem :: Eq a => a -> [a] -> Bool
Prelude> elem 'a' "abracadabra"
True
Prelude> elem 'a' "Julie"
False
```

In the first case, `'a'` is an element of `"abracadabra"`, so that evaluates to `True`, but in the second case, there is no `'a'` in `"Julie"`, so we get a `False` result. As you can see from the type signature, `elem` doesn't only work with characters and strings, but that's what we'll use it for here. Let's see if we can write a list comprehension to remove all the lowercase letters from a string. Here, our condition is that we only want to take `x` from our generator list when it meets the condition that it is an element of the list of capital letters:

```
Prelude> :{
Prelude| [x |
Prelude|  x <- "Three Letter Acronym",
Prelude|  elem x ['A'..'Z']]
Prelude| :}
"TLA"
```

Let's see if we can now generalize this into an acronym generator that will accept different strings as inputs, instead of forcing us to rewrite the whole list comprehension for every string we might want to feed it. We will do this by naming a function that will take one argument and use that as the generator string for our list comprehension. So, the function argument and the generator string will need to be the same thing:

[1] Reminder, pretend `Foldable` in the type of `elem` means it's a list until we cover `Foldable` later.

```
Prelude> :{
Prelude| let acro xs =
Prelude|      [x | x <- xs,
Prelude|           elem x ['A'..'Z']]
Prelude| :}
```

We use `xs` for our function argument to indicate to ourselves that it's a list, that the `x` is plural. It doesn't have to be; you could use a different variable there and obtain the same result. It is idiomatic to use a "plural" variable for list arguments, but it is not a requirement.

All right, so we have our `acro` function with which we can generate acronyms from any string:

```
Prelude> s = "Self"
Prelude> c = " Contained"
Prelude> u = " Underwater"
Prelude> b = " Breathing"
Prelude> a = " Apparatus"
Prelude> scuba = s ++ c ++ u ++ b ++ a
Prelude> acro scuba
"SCUBA"

Prelude> n = "National"
Prelude> a = " Aeronautics and"
Prelude> s = " Space"
Prelude> a' = " Administration"
Prelude> nasa = n ++ a ++ s ++ a'
Prelude> acro nasa
"NASA"
```

Given the above, what do you think this function would do:

```
Prelude> myString xs = [x | x <- xs, elem x "aeiou"]
```

Exercises: Square cube

Given the following:

```
Prelude> mySqr = [x^2 | x <- [1..5]]
Prelude> myCube = [y^3 | y <- [1..5]]
```

1. First write an expression that will make tuples of the outputs of `mySqr` and `myCube`.
2. Now, alter that expression so that it only uses the `x` and `y` values that are less than 50.
3. Apply another function to that list comprehension to determine how many tuples inhabit your output list.

9.8 Spines and non-strict evaluation

As we have seen, lists are a recursive series of cons cells `a : [a]` terminated by the empty list `[]`, but we want a way to visualize this structure in order to understand the ways lists get processed. When we talk about data structures in Haskell, particularly lists, sequences, and trees, we talk about them having a *spine*. This is the connective structure that ties the collection of values together. In the case of a list, the spine is usually textually represented by recursive cons (`:`) operators. Given the data `[1, 2]`, we get a list that looks like this:

```
1 : (2 : [])
```


The problem with the `1 : (2 : [])` representation we used earlier is that it makes it seem like the value 1 exists "before" the cons cell that contains it, but actually, the cons cells contain the values. Because of this and the way non-strict evaluation works, you can evaluate cons cells independently of what they contain. It is possible to evaluate only the spine of the list without evaluating individual values. It is also possible to evaluate only part of the spine of a list and not the rest of it.

Evaluation of the list in this representation proceeds down the spine. However, constructing the list (when that is necessary) proceeds *up* the spine. In the example above, then, we start with an infix operator, evaluate the arguments 1 and a new cons cell, and

proceed downward to the 2 and an empty list. When we build the list, it proceeds from the bottom of the list and up the spine, first putting the 2 in front of the empty list, then adding the 1 in front of the 2. Haskell's evaluation is non-strict, so the list isn't constructed until it's consumed. Nothing in the list is evaluated until it is forced. Until a value is consumed, there is a series of placeholders as a blueprint of the list that can be constructed when it's needed. We'll talk more about non-strictness soon.

We're going to bring ⊥, or bottom, back in the form of `undefined` in order to demonstrate some of the effects of non-strict evaluation. Here, we're going to use _ to syntactically signify values we are ignoring and not evaluating. The underscores represent the values contained by the cons cells. The spine is the recursive series of cons constructors signified by :, as you can see below:

```
  : <------|
 / \       |
_   : <----| This is the "spine"
   / \     |
  _   : <--|
     / \
    _  []
```

You'll see the term "spine" used in reference to data structures, such as trees, that aren't lists. In the case of a list, the spine is a linear succession of one cons cell wrapping another cons cell. With data structures like trees, which we will cover later, you'll see that the spine can be nodes that contain two or more nodes.

Using GHCi's :sprint command

We can use a special command in GHCi called `sprint` to print variables and see what has been evaluated already, with the underscore representing expressions that haven't been evaluated yet.

A warning: We always encourage you to experiment and explore for yourself after seeing the examples in this book, but `:sprint` has some behavioral quirks that can be a bit frustrating.

GHC Haskell has some opportunistic optimizations that introduce strictness to make code faster when it won't change how it evaluates.

Additionally, polymorphism means values like `Num a => a` are really waiting for a sort of argument that will make it concrete (this will be covered in more detail in a later chapter). To avoid this, you have to assign a more concrete type such as `Int` or `Double`, otherwise it stays unevaluated, _, in `:sprint`'s output. If you can keep these caveats to `:sprint`'s behavior in mind, it can be useful. Otherwise, if you find it confusing, don't sweat it, and wait for us to elaborate more deeply in the chapter on non-strictness.

Let's define a list using `enumFromTo`, which is tantamount to using syntax like `['a'..'z']`, then ask for the state of `blah` with respect to whether it has been evaluated:

```
Prelude> blah = enumFromTo 'a' 'z'
Prelude> :sprint blah
blah = _
```

The `blah = _` indicates that `blah` is totally unevaluated.

Next, we'll take one value from `blah` and then evaluate it by forcing GHCi to print the expression:

```
Prelude> take 1 blah
"a"
Prelude> :sprint blah
blah = 'a' : _
```

So, we've evaluated a cons cell, `:`, and the first value, `'a'`.

Then, we take two values and print them—which forces evaluation of the second cons cell and the second value:

```
Prelude> take 2 blah
"ab"
Prelude> :sprint blah
blah = 'a' : 'b' : _
```

Assuming this is a contiguous GHCi session, the first cons cell and value were already forced.

We can keep going with this, evaluating the list one value at a time:

```
Prelude> take 3 blah
"abc"
```

```
Prelude> :sprint blah
blah = 'a' : 'b' : 'c' : _
```

The `length` function is only strict in the spine, meaning it only forces evaluation of the spine of a list, not the values, something we can see if we try to find the length of a list of undefined values. But when we use `length` on `blah`, `:sprint` will behave as though we had forced evaluation of the values, as well:

```
Prelude> length blah
26
Prelude> :sprint blah
blah = "abcdefghijklmnopqrstuvwxyz"
```

That the individual characters were shown as evaluated and not exclusively the spine after getting the length of `blah` is one of the unfortunate aforementioned quirks of how GHCi evaluates code.

Spines are evaluated independently of values

Values in Haskell get reduced to weak head normal form by default. By "normal form," we mean that the expression is fully evaluated. "Weak head normal form" means the expression is only evaluated as far as is necessary to reach a data constructor.

Weak head normal form (WHNF) is a larger set and contains both the possibility that the expression is fully evaluated (normal form) and the possibility that the expression has been evaluated to the point of arriving at a data constructor or lambda awaiting an argument. For an expression in weak head normal form, further evaluation may be possible once another argument is provided. If no further inputs are possible, then it is still in WHNF but also in normal form (NF). We're going to explain this more fully later in the book in the chapter on non-strictness, when we show you how call-by-need works and the implications for Haskell. For now, we'll look at a few examples to get a sense of what might be going on.

Below, we list some expressions and whether they are in WHNF, NF, both, or neither:

```
(1, 2) -- WHNF & NF
```

This first example is in normal form and is fully evaluated. Anything in normal form is by definition also in weak head normal form, because weak head is an expression that is evaluated up to *at least* the first data constructor. Normal form exceeds that by requiring that all subexpressions be fully evaluated. Here the components of the value are the tuple data constructor and the values 1 and 2.

```
(1, 1 + 1)
```

This example is in WHNF but not NF. The + applied to its arguments could be evaluated but hasn't been yet.

```
\x -> x * 10 -- WHNF & NF
```

This anonymous function is in normal form, because while the * operator has been applied to two arguments of a sort, it cannot be reduced further until the outer `x -> ...` has been applied. With nothing further to reduce, it is in normal form.

```
"Papu" ++ "chon"
```

This string concatenation is in neither WHNF nor NF. This is because the outermost component of the expression is a function, ++, with arguments that are fully applied, but it hasn't been evaluated. Whereas, the following would be in WHNF but not NF:

```
(1, "Papu" ++ "chon")
```

When we define a list and also define all of its values, it is in NF, and all its values are known. There's nothing left to evaluate at that point, such as in the following example:

```
Prelude> num :: [Int]; num = [1, 2, 3]
Prelude> :sprint num
num = [1,2,3]
```

We can also construct a list through ranges or functions. In this case, the list is in WHNF but not NF. The compiler only evaluates the head or first node of the graph, but just the cons constructor, not the value or rest of the list it contains. We know there's a value of type `a` in the cons cell we haven't evaluated and a "rest of list" that might be

either the empty list `[]`, which ends the list, or another cons cell—we don't know which, because we haven't evaluated the next `[a]` value yet. We saw that above in the `:sprint` section, and you can see that evaluation of the first value does not force evaluation of the rest of the list:

```
Prelude> myNum :: [Int]; myNum = [1..10]
Prelude> :sprint myNum
myNum = _
Prelude> take 2 myNum
[1,2]
Prelude> :sprint myNum
myNum = 1 : 2 : _
```

This is an example of WHNF evaluation. It's weak head normal form, because the list has to be constructed by the range, and it's only going to evaluate as far as it has to. With `take 2`, we only need to evaluate the first two cons cells and the values they contain, which is why when we use `:sprint`, we only see `1 : 2 : _`. Evaluating to normal form would mean recursing through the entire list, forcing evaluation of not only the entire spine but also the values each cons cell contains.

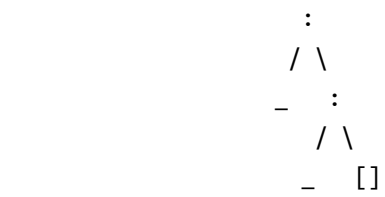
In these tree representations, evaluation or consumption of the list goes *down* the spine. The following is a representation of a list that isn't spine strict and is awaiting something to force the evaluation:

```
  :
 / \
_   _
```

By default, it stops here and never evaluates even the first cons cell unless it's forced to, as we saw.

However, functions that are spine strict can force complete evaluation of the spine of a list even if they don't force evaluation of each value. Pattern matching is strict by default, so pattern matching on cons cells can mean forcing spine strictness if your function doesn't stop recursing the list. It can evaluate the spine only or the spine as well as the values that inhabit each cons cell, depending on the context.

On the other hand, `length` is strict in the spine but not the values. If we define a list such as `[1, 2]`, using `length` on it would force evaluation of the entire spine without accompanying strictness in the values:



We can see this if we use `length`, but make one of the values bottom with the `undefined` value, and see what happens:

```
Prelude> x = [1, undefined]
Prelude> length x
2
```

The first value in the list is a number, but the second value is `undefined` and `length` doesn't make it crash. `length` measures the length of a list, which only requires recursing the spine and counting how many cons cells there are. We could define our own `length` function ourselves like so:

```
-- *Not* identical to the length
-- function in Prelude
length :: [a] -> Integer
length [] = 0
length (_:xs) = 1 + length xs
```

One thing to note is that we use _ to ignore the values in our arguments or that are part of a pattern match. In this case, we pattern match on the `:` data constructor but ignore the value that is the first argument. However, it's not a mere convention to bind references we don't care about on the left-hand side to _. You can't bind arguments to the name "_", because that symbol is part of the language. This is partly so the compiler knows for a certainty you won't ever evaluate something in that particular case. Currently, if you try using _ on the right-hand side of a definition, it'll think you're trying to refer to a hole.

We're only forcing the `:` constructors and the `[]` at the end in order to count the number of values contained by the list:

```
        :     <-|
       / \      | These got evaluated (forced)
|->   _   :   <-|
|       / \    |
|->    _   [] <-|
|
| These did not
```

However, `length` will throw an error on a bottom value if part of the spine itself is bottom:

```
Prelude> x = [1] ++ undefined ++ [3]
Prelude> x
[1*** Exception: Prelude.undefined
Prelude> length x
*** Exception: Prelude.undefined
```

Printing the list fails, although it gets as far as printing the first `[` and the first value, and attempting to get the length also fails, because it can't count undefined spine values.

It's possible to write functions that will force both the spine and the values. `sum` is an example, because in order to return a result at all, it must return the sum of all the values in a list.

We'll write our own `sum` function for the sake of demonstration:

```
mySum :: Num a => [a] -> a
mySum [] = 0
mySum (x : xs) = x + mySum xs
```

First, the `+` operator is strict in both of its arguments, so that will force evaluation of the values and the `mySum xs`. Therefore, `mySum` will keep recursing until it hits the empty list and must stop. Then it will start going back up the spine of the list, summing the inhabitants as it goes. It looks something like this (the zero represents our empty list):

```
Prelude> mySum [1..5]
1 + (2 + (3 + (4 + (5 + 0))))
1 + (2 + (3 + (4 + 5)))
1 + (2 + (3 + 9))
1 + (2 + 12)
1 + 14
15
```

We will be returning to this topic at various points in the book, because developing an intuition for Haskell's evaluation strategies takes time and practice. If you don't feel like you fully understand it at this point, that's OK. It's a complex topic, and it's better to approach it in stages.

Exercises: Bottom madness

Will it blow up?

Will the following expressions return a value or be ⊥?

1. `[x^y | x <- [1..5], y <- [2, undefined]]`
2. ```
   take 1 $
     [x^y | x <- [1..5], y <- [2, undefined]]
   ```
3. `sum [1, undefined, 3]`
4. `length [1, 2, undefined]`
5. `length $ [1, 2, 3] ++ undefined`
6. `take 1 $ filter even [1, 2, 3, undefined]`
7. `take 1 $ filter even [1, 3, undefined]`
8. `take 1 $ filter odd [1, 3, undefined]`
9. `take 2 $ filter odd [1, 3, undefined]`
10. `take 3 $ filter odd [1, 3, undefined]`
   ```

Intermission: Is it in normal form?

For each expression below, determine whether it's in:

1. Normal form, which implies weak head normal form.
2. Weak head normal form only.
3. Neither.

Remember that an expression cannot be in normal form *or* weak head normal form if the outermost part of the expression isn't a data constructor. It can't be in normal form if any part of the expression is unevaluated:

1. `[1, 2, 3, 4, 5]`
2. `1 : 2 : 3 : 4 : _`
3. `enumFromTo 1 10`
4. `length [1, 2, 3, 4, 5]`
5. `sum (enumFromTo 1 10)`
6. `['a'..'m'] ++ ['n'..'z']`
7. `(_, 'b')`

9.9 Transforming lists of values

We have already seen how we can make recursive functions with self-referential expressions. It's a useful tool and a core part of the logic of Haskell. In truth, in part because Haskell uses non-strict evaluation, we tend to use higher-order functions for transforming data rather than manually recursing over and over.

For example, one common thing you would want to do is return a list with a function applied uniformly to all values within that list. To do so, you need a function that is inherently recursive and can apply that function to each member of the list. For this purpose, we can use either the `map` or `fmap` functions. `map` can only be used with `[]`. `fmap` is defined in a type class named `Functor` and can be applied to data other than lists. We will learn more about `Functor` later; for now, we'll focus on the list usage. Here are some examples using `map` and `fmap`:

```
Prelude> map (+1) [1, 2, 3, 4]
[2,3,4,5]
Prelude> map (1-) [1, 2, 3, 4]
[0,-1,-2,-3]
Prelude> fmap (+1) [1, 2, 3, 4]
[2,3,4,5]
Prelude> fmap (2*) [1, 2, 3, 4]
[2,4,6,8]
Prelude> fmap id [1, 2, 3]
[1,2,3]
Prelude> map id [1, 2, 3]
[1,2,3]
```

The types of `map` and `fmap` respectively are:

```
map  ::             (a -> b) -> [a] -> [b]
fmap :: Functor f => (a -> b) -> f a -> f b
```

Let's look at how the types line up with a program, starting with `map`:

```
map :: (a -> b) -> [a] -> [b]


map (+1)
```

The `(a -> b)` becomes more specific and resolves to `Num a => a -> a`:

```
Prelude> :t map (+1)
map (+1) :: Num b => [b] -> [b]
```

Now, we see it will take one list of `Num` as an argument and return a list of `Num` as a result.

The type of `fmap` will behave similarly:

```
fmap :: Functor f => (a -> b) -> f a -> f b
-- notice the Functor type class constraint


fmap (+1)
-- again, (a -> b) is now more specific
```

It's a bit different from `map`, because the `Functor` type class includes more than lists:

```
Prelude> :t fmap (+1)
fmap (+1) :: (Num b, Functor f) => f b -> f b
```

Here's how `map` is defined in `base`:

```
map :: (a -> b) -> [a] -> [b]
map _  []     = []
-- [1] [2]     [3]
map f (x:xs)  = f x :  map f xs
-- [4] [5]     [6] [7] [8]
```

1. _ is used here to ignore the function argument, because we don't need it.

2. We are pattern matching on the `[]`, or empty list case, because the list datatype is a sum type with two cases, and we must handle both every time we pattern match or case on a list value.

3. We return the `[]` empty list value, because when there are no values, it's the only correct thing we can do. If you attempt to do anything else, the type checker will swat you.

4. We bind the function argument to the name `f`, as it merits no name more specific than this. `f` and `g` are common names for non-specific function values in Haskell. This is the function we are mapping over the list value with `map`.

5. We do not leave the entire list argument bound as a single name. Since we've already pattern matched the `[]` empty list case, we know there must be at least one value in the list. Here we pattern match into the `:`, the second data constructor of the list, which is a product. `x` is the single value of the cons product. `xs` is the rest of the list.

6. We apply our function `f` to the single value `x`. This part of the `map` function is what applies the function argument to the contents of the list.

7. We "cons" the value returned by the expression `f x` with the `:` operator onto the head of the result of applying `map` to the the rest of the list. Data is immutable in Haskell. When we map, we do not mutate the existing list but build a new list with the values that result from applying the function.

8. We call `map` itself applied to `f` and `xs`. This expression is the rest of the list with the function `f` applied to each value.

How do we write out what `map f` does? Note, this order of evaluation doesn't represent the proper non-strict evaluation order, but it does give an idea of what's going on:

```
map (+1) [1, 2, 3]
```

If we desugar the list syntax:

```
map (+1) (1 : (2 : (3 : [])))
```

`:` is defined as `infixr 5`, so the parentheses associate to the right. We aren't passing an empty list to `map`, so the second pattern match fires:

```
(+1) 1 :
  map (+1)
    (2 : (3 : []))
```

Apply `(+1)` to the next value, cons onto the result of mapping over the rest:

```
(+1) 1 :
  ((+1) 2 :
    (map (+1)
      (3 : [])))
```

This is the last time we'll trigger the second case of `map`:

```
(+1) 1 :
  ((+1) 2 :
    ((+1) 3 :
      (map (+1) [])))
```

Triggering the base case that handles the empty list and returns the empty list:

```
(+1) 1 :
  ((+1) 2 :
    ((+1) 3 : []))
```

Finishing the reduction of the expression:

```
2 : ((+1) 2 : ((+1) 3 : []))
2 : 3 : (+1) 3 : []
2 : 3 : 4 : [] == [2, 3, 4]
```

Using the syntactic sugar of list, here's an approximation of what `map` is doing for us:

```
map f [1, 2, 3] == [f 1, f 2, f 3]


map (+1) [1, 2, 3]
         [(+1) 1, (+1) 2, (+1) 3]
         [2, 3, 4]
```

Or using the spine syntax we introduced earlier:

```
   :
  / \
 1   :
    / \
   2   :
      / \
     3   []
```

Same spine syntax, but this time we're breaking down the expression `map (+1) [1, 2, 3]`:

```
       :
      / \
(+1) 1   :
        / \
  (+1) 2   :
          / \
    (+1) 3   []
```

As we mentioned above, these representations do not account for non-strict evaluation. Crucially, `map` doesn't traverse the whole list and apply the function immediately. The function is applied to the values you force out of the list one by one. We can see this by selectively leaving some values undefined:

```
Prelude> map (+1) [1, 2, 3]
[2,3,4]
```

The whole list is forced, because GHCi prints the list that results:

```
Prelude> (+1) undefined
*** Exception: Prelude.undefined

Prelude> (1, undefined)
(1,*** Exception: Prelude.undefined
Prelude> fst (1, undefined)
1

Prelude> badList = [1, 2, undefined]
Prelude> map (+1) badList
[2,3,*** Exception: Prelude.undefined

Prelude> take 2 $ map (+1) badList
[2,3]
```

In the final example, the `undefined` value is never forced, and there is no error, because we use `take 2` to request only the first two elements. With `map (+1)`, we only force as many values as cons cells we also force. We'll only force the values if we evaluate the result value in the list that the `map` function returns.

The significant part here is that strictness doesn't proceed only outside-in. We can have lazily evaluated code (e.g., `map`) wrapped around a strict core (e.g., `+`). In fact, we can choose to apply laziness and strictness in how we evaluate the spine or the leaves independently. A common mantra for performance sensitive code in Haskell is, "lazy in the spine, strict in the leaves." We'll cover this properly when we talk about non-strictness and data structures, although many Haskell users rarely worry about this.

You can use `map` and `fmap` with other functions and list types, as well. In this example, we use the `fst` function to return a list of the first element of each tuple in a list of tuples:

```
Prelude> map fst [(2, 3), (4, 5), (6, 7), (8, 9)]
[2,4,6,8]

Prelude> fmap fst [(2, 3), (4, 5), (6, 7), (8, 9)]
[2,4,6,8]
```

In this example, we map a partially applied `take` function:

```
Prelude> map (take 3) [[1..5], [1..5], [1..5]]
[[1,2,3],[1,2,3],[1,2,3]]
```

Next, we'll map an if-then-else expression over a list using an anonymous function. This list will find any value equal to 3, negate it, and then return the list:

```
Prelude> map (\x -> if x == 3 then (-x) else (x)) [1..10]
[1,2,-3,4,5,6,7,8,9,10]
```

At this point, you can try your hand at mapping different functions using this as a model. We recommend getting comfortable with mapping before moving on to the next chapter, on folding lists.

Exercises: More bottoms

As always, we encourage you to try figuring out the answers before you enter them into your REPL:

1. Will the following expression return a value or be ⊥?

   ```
   take 1 $ map (+1) [undefined, 2, 3]
   ```

2. Will the following expression return a value?

   ```
   take 1 $ map (+1) [1, undefined, 3]
   ```

3. Will the following expression return a value?

   ```
   take 2 $ map (+1) [1, undefined, 3]
   ```

4. What does the following mystery function do? What is its type? Describe it (to yourself or a loved one) in standard English and then test it out in the REPL to make sure you are correct:

```
itIsMystery xs =
  map (\x -> elem x "aeiou") xs
```

5. What will be the result of the following functions:

   a) `map (^2) [1..10]`

   b)
```
map minimum [[1..10], [10..20], [20..30]]
  -- n.b. minimum is not the same function
  -- as the min function that we used before
```

   c) `map sum [[1..5], [1..5], [1..5]]`

6. Back in Chapter 7, you wrote a function called `foldBool`. That function exists in a module known as `Data.Bool` and is called `bool`. Write a function that does the same (or similar, if you wish) as the `map` if-then-else function you saw above but uses `bool` instead of the if-then-else syntax. Your first step should be bringing the `bool` function into scope by typing `import Data.Bool` at your REPL prompt.

9.10 Filtering lists of values

When we talked about function composition in Chapter 7, we used a function called `filter` that takes a list as input and returns a new list consisting solely of the values in the input list that meet a certain condition, as in this example, which finds the even numbers of a list and returns a new list of those values:

```
Prelude> filter even [1..10]
[2,4,6,8,10]
```

Let's now take a closer look at `filter`. `filter` has the following definition:

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ []    = []
filter pred (x:xs)
  | pred x         = x : filter pred xs
  | otherwise      = filter pred xs
```

Filtering takes a function that returns a `Bool` value, maps that function over a list, and returns a new list of all the values that meet the condition. It's important to remind ourselves that this function, as we can see in the definition, *builds a new list* including values that meet the condition and excluding the ones that do not—it does not mutate the existing list.

We have seen how `filter` works with `odd` and `even` already. We have also seen one example along the lines of this:

```
Prelude> filter (== 'a') "abracadabra"
"aaaaa"
```

As you might suspect from what we've seen of HOFs, though, `filter` can handle many types of arguments. The following example does the same thing as `filter even` but with anonymous function syntax:

```
Prelude> xs = [1..20]
Prelude> filter (\x -> (rem x 2) == 0) xs
[2,4,6,8,10,12,14,16,18,20]
```

We covered list comprehensions earlier as a way of filtering lists, as well. Compare the following:

```
Prelude> filter (\x -> elem x "aeiou") "abracadabra"
"aaaaa"
Prelude> [x | x <- "abracadabra", elem x "aeiou"]
"aaaaa"
```

As they say, there's more than one way to skin a cat.

Again, we recommend at this point that you try writing some filter functions of your own to get comfortable with the pattern.

Exercises: Filtering

1. Given the above, how might we write a filter function that would give us all the multiples of 3 out of a list from 1–30?
2. Recalling what we learned about function composition, how could we compose the above function with the `length` function to tell us *how many* multiples of 3 there are between 1 and 30?
3. Next, we're going to work on removing all articles ("the," "a," and "an") from sentences. You want to get to something that works like this:

   ```
   Prelude> myFilter "the brown dog was a goof"
   ["brown","dog","was","goof"]
   ```

   You may recall that earlier in this chapter, we asked you to write a function that separates a string into a list of strings by separating them at spaces. That is a standard library function called `words`. You may consider starting this exercise by using `words` (or your own version, of course).

9.11 Zipping lists

Zipping lists together is a means of combining values from multiple lists into a single list. Related functions like `zipWith` allow you to use a combining function to produce a list of results from two lists.

First, let's look at `zip`:

```
Prelude> :t zip
zip :: [a] -> [b] -> [(a, b)]

Prelude> zip [1, 2, 3] [4, 5, 6]
[(1,4),(2,5),(3,6)]
```

One thing to note is that `zip` stops as soon as one of the lists runs out of values:

```
Prelude> zip [1, 2] [4, 5, 6]
[(1,4),(2,5)]
Prelude> zip [1, 2, 3] [4]
[(1,4)]
```

And will return an empty list if either of the lists is empty:

```
Prelude> zip [] [1..1000000000000000000]
[]
```

`zip` proceeds until the shortest list ends:

```
Prelude> zip ['a'] [1..1000000000000000000]
[('a',1)]
Prelude> zip [1..100] ['a'..'c']
[(1,'a'),(2,'b'),(3,'c')]
```

We can use `unzip` to recover the lists as they were before they were zipped:

```
Prelude> zip [1, 2, 3] [4, 5, 6]
[(1,4),(2,5),(3,6)]
Prelude> unzip $ zip [1, 2, 3] [4, 5, 6]
([1,2,3],[4,5,6])
Prelude> fst $ unzip $ zip [1, 2, 3] [4, 5, 6]
[1,2,3]
Prelude> snd $ unzip $ zip [1, 2, 3] [4, 5, 6]
[4,5,6]
```

Be aware that information can be lost in this process, because `zip` must stop on the shortest list:

```
Prelude> snd $ unzip $ zip [1, 2] [4, 5, 6]
[4,5]
```

We can also use `zipWith` to apply a function to the values of two lists in parallel:

```
zipWith :: (a -> b -> c)
--              [1]
        -> [a] -> [b] -> [c]
--         [2]    [3]    [4]
```

1. A function with two arguments. Notice how the type variables of the arguments and result align with the type variables in the lists.

2. The first input list.
3. The second input list.
4. The output list created by applying the function to the values in the input lists.

A brief demonstration of how `zipWith` works:

```
Prelude> zipWith (+) [1, 2, 3] [10, 11, 12]
[11,13,15]

Prelude> zipWith (*) [1, 2, 3] [10, 11, 12]
[10,22,36]

Prelude> zipWith (==) ['a'..'f'] ['a'..'m']
[True,True,True,True,True,True]

Prelude> xs = [10, 5, 34, 9]
Prelude> xs' = [6, 8, 12, 7]
Prelude> zipWith max xs xs'
[10,8,34,9]
```

Zipping exercises

1. Write your own version of `zip`, and ensure it behaves the same as the original:

```
zip :: [a] -> [b] -> [(a, b)]
zip = undefined
```

2. Do what you did for `zip` but now for `zipWith`:

```
zipWith :: (a -> b -> c)
        -> [a] -> [b] -> [c]
zipWith = undefined
```

3. Rewrite your `zip` in terms of the `zipWith` you wrote.

9.12 Chapter exercises

The first set of exercises here will mostly be review but will also introduce you to some new things. The second set is more conceptually challenging but does not use any syntax or concepts we haven't already studied. If you get stuck, it may help to flip back to a relevant section and review.

Data.Char

These first few exercises are straightforward but will introduce you to some new library functions and review some of what we've learned so far. Some of the functions we will use here are not standard in `Prelude` and so have to be imported from a module called `Data.Char`. You may do so in a source file (recommended) or at the `Prelude` prompt with the same phrase: `import Data.Char` (write that at the top of your source file). This brings into scope a bunch of new standard functions we can play with that operate on `Char` and `String` types.

1. Query the types of `isUpper` and `toUpper`.
2. Given the following behaviors, which would we use to write a function that filters all the uppercase letters out of a `String`? Write that function such that, given the input `"HbEfLrLxO"`, your function will return `"HELLO"`.

```
Prelude Data.Char> isUpper 'J'
True
Prelude Data.Char> toUpper 'j'
'J'
```

3. Write a function that will capitalize the first letter of a string and return the entire string. For example, if given the argument `"julie"`, it will return `"Julie"`.
4. Now make a new version of that function that is recursive, such that if you give it the input `"woot"`, it will holler back at you `"WOOT"`. The type signature won't change, but you will want to add a base case.

5. To do the final exercise in this section, we'll need another standard function for lists called `head`. Query the type of `head`, and experiment with it to see what it does. Now write a function that will capitalize the first letter of a `String` and return only that letter as the result.

6. Cool. Good work. Now rewrite it as a composed function. Then, for fun, rewrite it point-free.

Ciphers

We'll still be using `Data.Char` for this next exercise. You should save these exercises in a module called `Cipher`, because we'll be coming back to them in later chapters. You'll be writing a Caesar cipher for now, but we'll suggest some variations on the basic program in later chapters.

A Caesar cipher is a simple substitution cipher, in which each letter is replaced by the letter that is a fixed number of places down the alphabet from it. You will find variations on this all over the place—you can shift leftward or rightward, for any number of spaces. A rightward shift of three means that `'A'` will become `'D'`, and `'B'` will become `'E'`, for example. If you do a leftward shift of five, then `'a'` would become `'v'`, and so forth.

Your goal in this exercise is to write a basic Caesar cipher that shifts rightward. You can start by having the number of spaces to shift fixed, but it's more challenging to write a cipher that allows you to vary the number of shifts so that you can encode your secret messages differently each time.

There are Caesar ciphers written in Haskell all over the internet, but to maximize the likelihood that you can write yours without peeking at those, we'll provide a couple of tips. When yours is working the way you want it to, we would encourage you to then look around and compare your solution to others out there.

The first lines of your text file should look like this:

```
module Cipher where

import Data.Char
```

`Data.Char` includes two functions called `ord` and `chr` that can be used to associate a `Char` with its `Int` representation in the Unicode system and vice versa:

```
*Cipher> :t chr
chr :: Int -> Char
*Cipher> :t ord
ord :: Char -> Int
```

Using these functions is optional; there are other ways you can proceed with shifting, but using `chr` and `ord` might simplify the process a bit.

You want your shift to wrap back around to the beginning of the alphabet, so that if you have a rightward shift of three from `'z'`, you end up back at `'c'` and not somewhere in the vast Unicode hinterlands. Depending on how you've set things up, this might be a bit tricky. Consider starting from a base character (e.g., `'a'`) and using `mod` to ensure you're only shifting over the 26 standard characters of the English alphabet.

You should include an `unCaesar` function that will decipher your text, as well. In a later chapter, we will test it.

Writing your own standard functions

Below are the outlines of some standard functions. The goal here is to write your own versions of these to gain a deeper understanding of recursion over lists and how to make functions flexible enough to accept a variety of inputs. You could figure out how to look up the answers, but you won't do that, because you know you'd only be cheating yourself out of the knowledge. Right?

Let's look at an example of what we're after here. The `and`[2] function takes a list of `Bool` values and returns `True` if and only if no values in the list are `False`. Here's how you might write your own version of it:

[2]Note that if you're using GHC 7.10 or newer, the functions `and`, `any`, and `all` have been abstracted from being usable only with lists to being usable with any datatype that has an instance of the type class `Foldable`. It still works with lists, the same as it did before. Proceed assured that we'll cover this later.

```
-- direct recursion, not using (&&)
myAnd :: [Bool] -> Bool
myAnd [] = True
myAnd (x:xs) =
  if x == False
  then False
  else myAnd xs

-- direct recursion, using (&&)
myAnd :: [Bool] -> Bool
myAnd [] = True
myAnd (x:xs) = x && myAnd xs
```

And now the fun begins:

1. `myOr` returns `True` if any `Bool` in the list is `True`:

   ```
   myOr :: [Bool] -> Bool
   myOr = undefined
   ```

2. `myAny` returns `True` if `a -> Bool` applied to any of the values in the list returns `True`:

   ```
   myAny :: (a -> Bool) -> [a] -> Bool
   myAny = undefined
   ```

   Example for validating `myAny`:

   ```
   Prelude> myAny even [1, 3, 5]
   False
   Prelude> myAny odd [1, 3, 5]
   True
   ```

3. After you write the recursive `myElem`, write another version that uses `any`. The built-in version of `elem` in GHC 7.10 and newer has a type that uses `Foldable` instead of the list type, specifically. You can ignore that and write the concrete version that works only for lists:

   ```
   myElem :: Eq a => a -> [a] -> Bool
   ```

```
Prelude> myElem 1 [1..10]
True
Prelude> myElem 1 [2..10]
False
```

4. Implement `myReverse`:

```
myReverse :: [a] -> [a]
myReverse = undefined

Prelude> myReverse "blah"
"halb"
Prelude> myReverse [1..5]
[5,4,3,2,1]
```

5. `squish` flattens a list of lists into a list:

```
squish :: [[a]] -> [a]
squish = undefined
```

6. `squishMap` maps a function over a list and concatenates the results:

```
squishMap :: (a -> [b]) -> [a] -> [b]
squishMap = undefined

Prelude> squishMap (\x -> [1, x, 3]) [2]
[1,2,3]
Prelude> squishMap (\x -> "WO "++[x]++" HOO ") "123"
"WO 1 HOO WO 2 HOO WO 3 HOO "
```

7. `squishAgain` flattens a list of lists into a list. This time, re-use the `squishMap` function:

```
squishAgain :: [[a]] -> [a]
squishAgain = undefined
```

8. `myMaximumBy` takes a comparison function and a list and returns the greatest element of the list based on the last value that the comparison returns `GT` for. If you import `maximumBy` from `Data.List`, you'll see that the type is:

```
Foldable t => (a -> a -> Ordering)
  -> t a -> a
```

Rather than:

```
(a -> a -> Ordering) -> [a] -> a
```

```
myMaximumBy :: (a -> a -> Ordering)
            -> [a] -> a
myMaximumBy = undefined

Prelude> xs = [1, 53, 9001, 10]
Prelude> myMaximumBy compare xs
9001
```

9. `myMinimumBy` takes a comparison function and a list and returns the least element of the list based on the last value that the comparison returns `LT` for:

```
myMinimumBy :: (a -> a -> Ordering)
            -> [a] -> a
myMinimumBy = undefined

Prelude> xs = [1, 53, 9001, 10]
Prelude> myMinimumBy compare xs
1
```

10. Using the `myMinimumBy` and `myMaximumBy` functions, write your own versions of `maximum` and `minimum`. If you have GHC 7.10 or newer, you'll see a type constructor that wants a `Foldable` instance instead of a list, as has been the case for many functions so far:

```
myMaximum :: (Ord a) => [a] -> a
myMaximum = undefined

myMinimum :: (Ord a) => [a] -> a
myMinimum = undefined
```

9.13 Definitions

1. In type theory, a *product type* is a type made of a set of types compounded over each other. In Haskell, we represent products using tuples or data constructors with more than one argument. The "compounding" is from each type argument to the data constructor representing a value that coexists with all the other values simultaneously. Products of types represent a conjunction, "and," of those types. If you have a product of `Bool` and `Int`, your terms will *each* contain a `Bool` *and* an `Int` value.

2. In type theory, a *sum type* of two types is a type whose terms are terms in either type, but not simultaneously. In Haskell, sum types are represented using the pipe, `|`, in a datatype definition. Sums of types represent a disjunction, "or," of those types. If you have a sum of `Bool` and `Int`, your terms will be *either* a `Bool` value *or* an `Int` value.

3. *Cons* is ordinarily used as a verb to signify that a list value has been created by *cons*'ing a value onto the head of another list value. In Haskell, `:` is the cons operator for the list type. It is a data constructor defined in the list datatype:

```
   1 : [2, 3]
-- [a]   [b]

 [1, 2, 3]
--   [c]

 (:) :: a -> [a] -> [a]
--     [d]   [e]   [f]
```

   a) The number 1, the value we are cons'ing.
   b) A list of the number 2 followed by the number 3.
   c) The final result of cons'ing 1 onto `[2, 3]`.
   d) The type variable `a` corresponds to 1, the value we cons'ed onto the list value.
   e) The first occurrence of the type `[a]` in the cons operator's type corresponds to the second and final argument that `:` accepts, which is `[2, 3]`.

f) The second and final occurrence of the type `[a]` in the cons operator's type corresponds to the final result, `[1, 2, 3]`.

4. A *cons cell* is a data constructor and a product of the types `a` and `[a]` as defined in the list datatype. Because it references the list type constructor itself in the second argument, it allows for the nesting of multiple cons cells, possibly indefinitely with the use of recursive functions, for representing an indefinite number of values in a series:

```
data [] a = []  |  a : [a]
--                 ^ cons operator
```

We can also define it ourselves:

```
data List a = Nil | Cons a (List a)
```

And then create a list using our own list type:

```
Cons 1 (Cons 2 (Cons 3 Nil))
```

Here, `(Cons 1 ...)`, `(Cons 2 ...)`, and `(Cons 3 Nil)` are all individual cons cells in the list `[1, 2, 3]`.

5. The *spine* is a way to refer to the structure that glues a collection of values together. In the list datatype, it is formed by the recursive nesting of cons cells. The spine is, in essence, the structure of the collection that *isn't* the values contained therein. Often, the term *spine* is used in reference to lists, but it applies in the case of data structures shaped like trees, as well. Given the list `[1, 2, 3]`:

```
1 : --------| The nested cons operators
  (2 : -----| here represent the spine.
     (3 : --|
         []))
```

Blanking the irrelevant values out:

```
_ : ----------|
  (_ : -------|
     (_ : ----> Spine
         []))
```

9.14 Follow-up resources

1. `Data.List` documentation for the `base` library.
   `http://hackage.haskell.org/package/base/docs/Data-List.html`
2. Haskell Wiki. *Ninety-Nine Haskell problems*.
   `https://wiki.haskell.org/H-99:_Ninety-Nine_Haskell_Problems`

Chapter 10

Folding Lists

> The explicit teaching of thinking is no trivial task, but who said that the teaching of programming is? In our terminology, the more explicitly thinking is taught, the more of a scientist the programmer will become.
>
> Edsger Dijkstra

10.1 Folds

Folding is a concept that extends in usefulness and importance beyond lists, but lists are often how they are introduced. Folds as a general concept are called catamorphisms. You're familiar with the root "morphism" from polymorphism. "Cata-" means "down" or "against," as in "catacombs." Catamorphisms are a means of deconstructing data. If the spine of a list is the structure of a list, then a fold is what can reduce that structure.[1]

This chapter is a thorough look at the topic of folding lists in Haskell. We will:

- Explain what folds are and how they work.
- Detail the evaluation processes of folds.
- Walk through writing folding functions.
- Introduce scans, functions that are related to folds.

10.2 Bringing you into the fold

Let's start with a quick look at `foldr`, short for "fold right." This is the fold you'll most often want to use with lists. The following type signature may look a little hairy, but let's compare it to what we know about mapping. Note that the type of `foldr` changed with GHC 7.10:

```
-- GHC 7.8 and older
foldr :: (a -> b -> b) -> b -> [a] -> b

-- GHC 7.10 and newer
foldr :: Foldable t
      => (a -> b -> b)
      -> b
      -> t a
      -> b
```

Lined up next to each other:

[1] Note that a catamorphism *can* break down the structure but that structure might be rebuilt, so to speak, during evaluation. That is, folds can return lists as results.

```
foldr :: Foldable t =>
        (a -> b -> b) -> b -> t  a -> b
foldr :: (a -> b -> b) -> b -> [] a -> b
```

For now, all you need to know is that GHC 7.10 abstracted out the list-specific part of folding into a type class that lets you reuse the same folding functions for any datatype that can be folded—not just lists. We can even recover the more concrete type, because we can always make a type more concrete, but never more generic:

```
Prelude> :{
Prelude| let listFoldr :: (a -> b -> b)
Prelude|               -> b
Prelude|               -> [] a
Prelude|               -> b
Prelude|     listFoldr = foldr
Prelude| :}
Prelude> :t listFoldr
listFoldr :: (a -> b -> b) -> b -> [a] -> b
```

Now, let's notice a parallel between `map` and `foldr`:

```
foldr :: (a -> b -> b) -> b -> [a] -> b

-- Remember how map works?
map   :: (a -> b) -> [a] -> [b]
map (+1) 1  :       2  :       3  : []

    (+1) 1  :  (+1) 2  :  (+1) 3  : []

-- Given the list
foldr (+) 0 (1  :   2  :   3  :  [])

            1  +  (2  +  (3  +   0))
```

Where `map` applies a function to each member of a list and returns a list, a fold replaces the cons constructors with the function and reduces the list.

10.3 Recursive patterns

Let's revisit `sum`:

```
Prelude> sum [1, 5, 10]
16
```

As we've seen, it takes a list, adds the elements together, and returns a single result. You might think of it as similar to the `map` functions we've looked at, except that it's mapping `+` over the list, replacing the cons operators themselves, and returning a single result, instead of mapping, for example, `(+1)` into each cons cell and returning a whole list of results back to us. This has the effect of both mapping an operator over a list and also reducing the list. In a previous section, we wrote `sum` in terms of recursion:

```
sum :: [Integer] -> Integer
sum []     = 0
sum (x:xs) = x + sum xs
```

And if we bring back our `length` function from earlier:

```
length :: [a] -> Integer
length []     = 0
length (_:xs) = 1 + length xs
```

Do you see some structural similarity? What if you look at `product` and `concat`, as well?

```
product :: [Integer] -> Integer
product []     = 1
product (x:xs) = x * product xs

concat :: [[a]] -> [a]
concat []     = []
concat (x:xs) = x ++ concat xs
```

In each case, the base case is the identity for that function. So the identity for `sum`, `length`, `product`, and `concat`, respectively, are 0, 0, 1, and `[]`. When we do addition, adding zero gives us the same result as our initial value: 1 + 0 = 1. But when we do multiplication,

it's multiplying by 1 that gives us the identity: $2 \times 1 = 2$. With list concatenation in Haskell, the identity is the empty list `[]`, such that `[1, 2, 3] ++ [] == [1, 2, 3]`.

Also, each of them has a main function with a recursive pattern that associates to the right. The head of the list gets evaluated, set aside, and then the function moves to the right, evaluates the next head, and so on.

10.4 Fold right

We call `foldr` the "right fold," because the fold is right associative, that is, it associates to the right. This is syntactically reflected in a straightforward definition of `foldr`, as well:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)
```

The similarities between this and the recursive patterns we saw above should be clear. The "rest of the fold," `(foldr f z xs)`, is an argument to the function `f` we're folding with. The `z` is the zero of our fold. It provides a fallback value for the empty list case and a second argument to begin our fold with. The zero is often the identity for whatever function we're folding with, such as 0 for `+` and 1 for `*`.

How foldr evaluates

We're going to rejigger our definition of `foldr` a little bit. It won't change the semantics, but it'll make it easier to write out what's happening:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z xs =
  case xs of
    []     -> z
    (x:xs) -> f x (foldr f z xs)
```

Here, we see how the right fold associates to the right. This will reduce like the `sum` example from earlier:

```
foldr (+) 0 [1, 2, 3]
```

When we reduce that fold, the first step is substituting `xs` in our case expression:

```
foldr (+) 0 [1, 2, 3] =
  case [1, 2, 3] of
    ...
```

Which case of the expression matches?

```
foldr (+) 0 [1, 2, 3] =
  case [1, 2, 3] of
    []     -> 0
    (x:xs) ->
      f x (foldr f z xs) -- <--- this one
```

What are `f`, `x`, `xs`, and `z` in that branch of the case?

```
foldr (+) 0 [1, 2, 3] =
  case [1, 2, 3] of
    []           -> 0
    (1 : [2, 3]) ->
      (+) 1 (foldr (+) 0 [2, 3])
```

Critically, we're going to expand `(foldr (+) 0 [2, 3])` only because `+` is strict in both of its arguments, so it forces the next iteration. We could have a function that doesn't continually force the rest of the fold. If it were to stop on the first case here, then it would have returned the value 1. One such function is `const`, which always returns the first argument. We'll show you how that behaves in a bit. Our next recursion is `(foldr (+) 0 [2, 3])`:

```
foldr (+) 0 [2, 3] =
  case [2, 3] of
  []        ->
    0 -- this doesn't match again
  (2 : [3]) -> (+) 2 (foldr (+) 0 [3])
```

There is a `(+) 1` implicitly wrapped around this continuation of the recursive fold. `+` is not only strict in both of its arguments, but it's *unconditionally* so, so we're going to proceed to the next recursion of `foldr`. Note that the function calls bounce between our folding function `f` and `foldr`. This bouncing back and forth gives more control to the folding function. A hypothetical folding function, such as `const`, which doesn't need the second argument, has the opportunity to do less work by not evaluating its second argument, which is "more of the fold."

`(+) 1 ((+) 2 ...)` is implicitly wrapped around this next step of the recursive fold:

```
foldr (+) 0 [3] =
  case [3] of
  []       ->
    0 -- this doesn't match again
  (3 : []) -> (+) 3 (foldr (+) 0 [])
```

We're going to ask for more `foldr` one last time. We have, again, `(+) 1 ((+) 2 ((+) 3 ...))` implicitly wrapped around this final step of the recursive fold. Finally, we hit our base case:

```
foldr (+) 0 [] =
  case [] of
    []       ->
      0    -- <-- This one finally matches
    -- ignore the other case,
    -- it doesn't happen
```

So one way to think about the way Haskell evaluates is that it's like a text rewriting system. Our expression has thus far rewritten itself from:

```
foldr (+) 0 [1, 2, 3]
```

Into:

```
(+) 1 ((+) 2 ((+) 3 0))
```

If you wanted to clean it up a bit without changing how it evaluates, you could make it the following:

```
1 + (2 + (3 + 0))
```

As in arithmetic, we evaluate innermost parentheses first:

```
1 + (2 + (3 + 0))

1 + (2 + 3)

1 + 5

6
```

And now we're done, with the result of 6.

We can also use a trick popularized by some helpful users in the Haskell IRC community to see how the fold associates:[2]

```
xs = map show [1..5]
y = foldr (\x y -> concat
    ["(",x,"+",y,")"]) "0" xs
```

When we call `y` in the REPL, we can see how `foldr` evaluates:

```
Prelude> y
"(1+(2+(3+(4+(5+0)))))"
```

One initially non-obvious aspect of folding is that it happens in two stages, traversal and folding. Traversal is the stage in which the fold recurses over the spine. Folding refers to the evaluation or reduction of the folding function applied to the values. All folds recurse over the spine in the same direction; the difference between left folds and right folds is in the association, or parenthesization, of the folding function and, thus, in which direction the folding or reduction proceeds.

With `foldr`, the rest of our fold is an argument to the function we're folding with:

```
foldr f z (x:xs) = f x (foldr f z xs)
--                     ^------------^
--                     rest of the fold
```

[2] Idea borrowed from Cale Gibbard from the `#haskell` Freenode IRC channel and on the Haskell Wiki `https://wiki.haskell.org/Fold#Examples`.

Given this two-stage process and non-strict evaluation, if `f` doesn't evaluate its second argument (the rest of the fold), no more of the spine will be forced. One of the consequences of this is that `foldr` can avoid evaluating not only some or all of the values in the list, but some or all of the list's *spine*, as well! For this reason, `foldr` can be used with lists that are potentially infinite. For example, compare the following sets of results (recall that `+` will unconditionally evaluate the entire spine and all of the values):

```
Prelude> foldr (+) 0 [1..5]
15
```

While you cannot use `foldr` with addition on an infinite list, you can use functions that are not strict in both arguments and therefore do not require evaluation of every value in order to return a result. The function `myAny`, for example, can return a `True` result as soon as it finds one `True`:

```
myAny :: (a -> Bool) -> [a] -> Bool
myAny f xs =
  foldr (\x b -> f x || b) False xs
```

The following should work despite being an infinite list:

```
Prelude> myAny even [1..]
True
```

The following, however, will never finish evaluating, because it's always an odd number:

```
Prelude> myAny even (repeat 1)
```

Another term we use—and that we've seen before—for this never-ending evaluation is *bottom* or `undefined`. There's no guarantee that a fold of an infinite list will finish evaluating even if you use `foldr`, as it often depends on the input data and the fold function you supply to operate on it. Let us consider some more examples with a less inconvenient bottom:

```
Prelude> u = undefined

-- here, we give an undefined value
Prelude> foldr (+) 0 [1, 2, 3, 4, u]
*** Exception: Prelude.undefined

Prelude> xs = take 4 [1, 2, 3, 4, u]
Prelude> foldr (+) 0 xs
10

-- here, undefined is part of the spine
Prelude> xs = [1, 2, 3, 4] ++ u
Prelude> foldr (+) 0 xs
*** Exception: Prelude.undefined
Prelude> xs = take 4 ([1, 2, 3, 4]++u)
Prelude> foldr (+) 0 xs
10
```

By taking only the first four elements, we stop the recursive folding process after the fourth value, so our addition function does not run into bottom, and that works whether `undefined` is one of the values or part of the spine.

The `length` function behaves differently; it evaluates the spine unconditionally but not the values:

```
Prelude> length [1, 2, 3, 4, undefined]
5
Prelude> length ([1, 2, 3, 4] ++ undefined)
*** Exception: Prelude.undefined
```

However, if we drop the part of the spine that includes the bottom before we use `length`, we can get an expression that works:

```
Prelude> xs = [1, 2, 3, 4] ++ undefined
Prelude> length (take 4 xs)
4
```

The `take` function is non-strict like everything else you've seen so far, and in this case, it only returns as much list as you ask for. The difference in what it does is that it *stops* returning elements from a list when it hits the given length limit. Consider this:

```
Prelude> xs = [1, 2] ++ undefined
Prelude> length $ take 2 $ take 4 xs
2
```

It doesn't matter that `take 4` could have hit the bottom! Nothing forced it to because of the `take 2` between it and `length`.

Now that we've seen how the recursive second argument to `foldr`'s folding function works, let's consider the first argument:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)
--                  ^-- first argument
```

The first argument, noted above, involves a pattern match that is strict by default—the `f` only applies to `x` if there is an `x` value and not just an empty list. This means that `foldr` must force an initial cons cell in order to discriminate between the `[]` and the `(x:xs)` cases, so the first cons cell *cannot* be undefined.

Now, we're going to try something unusual to demonstrate that the first bit of the spine must be evaluated by `foldr`. We have a somewhat silly, anonymous function that will ignore all of its arguments and return a value of 9001. We're using it with `foldr`, because it will never force evaluation of any of its arguments, so we can have a bottom as a value or as part of the spine, and it will not force an evaluation:

```
Prelude> foldr (\_ _ -> 9001) 0 [1..5]
9001
Prelude> xs = [1, 2, 3, undefined]
Prelude> foldr (\_ _ -> 9001) 0 xs
9001
Prelude> xs = [1, 2, 3] ++ undefined
Prelude> foldr (\_ _ -> 9001) 0 xs
9001
```

Everything is fine unless the first cons cell of the spine is bottom:

```
Prelude> foldr (\_ _ -> 9001) 0 undefined
*** Exception: Prelude.undefined
```

```
Prelude> xs = [1, undefined]
Prelude> foldr (\_ _ -> 9001) 0 xs
9001
Prelude> xs = [undefined, undefined]
Prelude> foldr (\_ _ -> 9001) 0 xs
9001
```

The final two examples work, because it isn't the first cons cell that is bottom—the `undefined` values are inside the cons cells, not in the spine itself. Put differently, the cons cells *contain* bottom values but are not themselves bottom. We will experiment later with non-strictness and strictness to see how they affect the way our programs evaluate.

Traversing the rest of the spine doesn't occur unless the function asks for the result of having folded the rest of the list. In the following examples, we don't force traversal of the spine, because `const` throws away its second argument, which is the rest of the fold:

```
-- reminder:
-- const :: a -> b -> a
-- const x _ =  x

Prelude> const 1 2
1
Prelude> const 2 1
2
Prelude> foldr const 0 [1..5]
1
Prelude> foldr const 0 [1,undefined]
1
Prelude> foldr const 0 ([1,2] ++ undefined)
1
Prelude> foldr const 0 [undefined,2]
*** Exception: Prelude.undefined
```

Now that we've seen how `foldr` evaluates, we're going to look at `foldl` before we move on to learning how to write and use folds.

10.5 Fold left

Because of the way lists work, folds must first recurse over the spine of the list from beginning to end. Left folds traverse the spine in the same direction as right folds, but their folding process is left associative and proceeds in the opposite direction as that of `foldr`.

Here's a simple definition of `foldl`. Note that to see the same type for `foldl` in your GHCi REPL, you will need to import `Data.List` for the same reason as for `foldr`:

```
-- again, different type in
-- GHC 7.10 and newer.

foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f acc []     =  acc
foldl f acc (x:xs) =  foldl f (f acc x) xs

foldl :: (b -> a -> b) -> b -> [a] -> b

-- Given the list
foldl (+) 0 (1  :   2  :   3  :  [])

-- foldl associates like this
       ((0     + 1)   +  2)  +  3
```

We can also use the same trick we used to see the associativity of `foldr` to see the associativity of `foldl`:

```
Prelude> conc = concat
Prelude> f x y = conc ["(",x,"+",y,")"]

Prelude> foldl f "0" (map show [1..5])
"(((((0+1)+2)+3)+4)+5)"
```

We can see from this that `foldl` begins its reduction process by adding the `acc` (accumulator) value to the head of the list, whereas `foldr` adds it to the final element of the list, first.

We can also use functions called *scans* to see how folds evaluate. Scans are similar to folds but return a list of all the intermediate stages

of the fold. We can compare `scanr` and `scanl` to their accompanying folds to see the difference in evaluation:

```
Prelude> foldr (+) 0 [1..5]
15
Prelude> scanr (+) 0 [1..5]
[15,14,12,9,5,0]

Prelude> foldl (+) 0 [1..5]
15
Prelude> scanl (+) 0 [1..5]
[0,1,3,6,10,15]
```

The relationship between scans and folds is as follows:

```
last (scanl f z xs) = foldl f z xs
head (scanr f z xs) = foldr f z xs
```

Each fold will return the same result for this operation, but we can see from the scans that they arrive at that result in a different order, due to the different associativity. We'll talk more about scans later.

Associativity and folding

Next, we'll take a closer look at some of the effects of the associativity of `foldl`. As we've said, both folds traverse the spine in the same direction. What's different is the associativity of the evaluation.

The fundamental way to think about evaluation in Haskell is as substitution. When we use a right fold on a list with the function `f` and start value `z`, we're, in a sense, replacing the cons constructors with our folding function and the empty list constructor with our start value `z`:

```
[1..3] == 1 : 2 : 3 : []

foldr f z [1, 2, 3]
1 `f` (foldr f z [2, 3])
1 `f` (2 `f` (foldr f z [3]))
1 `f` (2 `f` (3 `f` (foldr f z [])))
1 `f` (2 `f` (3 `f` z))
```

Furthermore, lazy evaluation lets our functions, rather than the ambient semantics of the language, dictate in which order things get evaluated. Because of this, the *parentheses are real*. In the above, the `3 `f` z` pairing gets evaluated first, because it's in the innermost parentheses. Right folds have to traverse the list outside-in, but the folding itself starts from the end of the list.

It's hard to see this with arithmetic functions that are associative, such as addition, but it's an important point to understand, so we'll run through some different examples. Let's start by using an arithmetic operation that isn't associative:

```
Prelude> foldr (^) 2 [1..3]
1
Prelude> foldl (^) 2 [1..3]
64
```

This time we can see clearly that we get different results, and that difference results from the way the functions associate. Here's a breakdown:

```
-- If you want to follow along,
-- use paper and not the REPL.
foldr (^) 2 [1..3]
(1 ^ (2 ^ (3 ^ 2)))
(1 ^ (2 ^ 9))
1 ^ 512
1
```

Contrast that with this:

```
foldl (^) 2 [1..3]
((2 ^ 1) ^ 2) ^ 3
(2 ^ 2) ^ 3
4 ^ 3
64
```

In this next set of comparisons, we will demonstrate the effect of associativity on argument order by folding the same list into a new list, like this:

```
Prelude> foldr (:) [] [1..3]
[1,2,3]
Prelude> foldl (flip (:)) [] [1..3]
[3,2,1]
```

We must use `flip` with `foldl`. Let's examine why.

Like a right fold, a left fold cannot perform magic and go to the end of the list instantly; it must start from the beginning of the list. However, the parentheses dictate how our code evaluates. The type of the argument to the folding function changes in addition to the associativity:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
--       [1]  [2]  [3]
foldl :: (b -> a -> b) -> b -> [a] -> b
--       [4]  [5]  [6]
```

1. The parameter of type `a` represents one of the list element arguments the folding function of `foldr` is applied to.
2. The parameter of type `b` will either be the start value or the result of the fold accumulated so far, depending on how far you are into the fold.
3. The final result of having combined the list element and the start value or fold so far to compute the fold.
4. The start value or fold accumulated so far is the first argument to `foldl`'s folding function.
5. The list element is the second argument to `foldl`'s folding function.
6. The final result of `foldl`'s fold function is of type `b`, like that of `foldr`.

The type of `:` requires that a value be the first argument and a list be the second argument:

```
(:) :: a -> [a] -> [a]
```

So the value is prepended, or "cons'ed onto," the front of that list.

In the following examples, the tilde means "is equivalent or equal to." If we write a right fold that has the cons constructor as our `f` and the empty list as our `z`, we get:

```
-- foldr f z [1, 2, 3]
-- f ~ (:); z ~ []
-- Run it in your REPL. It'll return True.
   foldr (:) [] (1 : 2 : 3 : [])
== 1 : (2 : (3 : []))
```

The cons'ing process for `foldr` matches the type signature for the `:` operator. It also reproduces the same list, because we're replacing the cons constructors with cons constructors and the null list with null list. However, for it to be identical, it also has to be right associative.

Doing the same thing with `foldl` does not produce the same result. When using `foldl`, the result we've accumulated so far is the first argument instead of the list element. This is the opposite of what `:` expects if we're accumulating a list. Trying to fold the identity of the list as above but with `foldl` would give us a type error, because the reconstructing process for `foldl` would look like this:

```
foldl f z [1, 2, 3]
-- f ~ (:); z ~ []
-- (((z `f` 1) `f` 2) `f` 3)
(([] : 1) : 2) : 3)
```

That won't work, because the `z` is an empty list and the `f` is cons, so we have the order of arguments backwards for cons. Enter `flip`, which takes backwards arguments and turns that frown upside-down. It will flip each set of arguments around for us, like this:

```
foldl f z [1, 2, 3]
-- f ~ (flip (:)); z ~ []
-- (((z `f` 1) `f` 2) `f` 3)
f = flip (:)
((([] `f` 1) `f` 2) `f` 3)
 (([1] `f` 2) `f` 3)
  ([2, 1] `f` 3)
   [3, 2, 1]
```

Even when we've satisfied the types by flipping things around, the left-associating nature of `foldl` leads to a different result from that of `foldr`.

For the next set of comparisons, we're going to use a function called `const` that takes two arguments and always returns the first one. When we fold `const` over a list, it will take as its first pair of arguments the `acc` value and a value from the list—which value it takes first depends on which type of fold it is. We'll show you how it evaluates for the first example:

```
Prelude> foldr const 0 [1..5]
(const 1 _)
1
```

Since `const` doesn't evaluate its second argument, the rest of the fold is never evaluated. The underscore represents the rest of the unevaluated fold. Now, let's look at the effect of flipping the arguments. The 0 result is because zero is our accumulator value here, so it's the first (or last) value of the list:

```
Prelude> foldr (flip const) 0 [1..5]
0
```

Next, let's look at what happens when we use the same functions but this time with `foldl`. Take a few moments to understand the evaluation process that leads to these results:

```
Prelude> foldl (flip const) 0 [1..5]
5
Prelude> foldl const 0 [1..5]
0
```

This is the effect of left associativity. The spine traversal happens in the same order in a left or right fold—it must, because of the way lists are defined. Depending on your folding function, however, a left fold can lead to a different result than a right fold of the same list.

Exercises: Understanding folds

1. `foldr (*) 1 [1..5]`

Will return the same result as which of the following?

a) `flip (*) 1 [1..5]`

b) `foldl (flip (*)) 1 [1..5]`

c) `foldl (*) 1 [1..5]`

2. Write out the evaluation steps for:

```
foldl (flip (*)) 1 [1..3]
```

3. One difference between `foldr` and `foldl` is:

a) `foldr`, but not `foldl`, traverses the spine of a list from right to left.

b) `foldr`, but not `foldl`, always forces the rest of the fold.

c) `foldr`, but not `foldl`, associates to the right.

d) `foldr`, but not `foldl`, is recursive.

4. Folds are catamorphisms, which means they are generally used to:

a) Reduce structure.

b) Expand structure.

c) Render you catatonic.

d) Generate infinite data structures.

5. The following are simple folds very similar to what you've already seen, but each has at least one error. Please fix and test them in your REPL:

a) `foldr (++) ["woot", "WOOT", "woot"]`

b) `foldr max [] "fear is the little death"`

c) `foldr and True [False, True]`

d) This one is more subtle than the previous. Can it ever return a different answer?

```
foldr (||) True [False, True]
```

e) `foldl ((++) . show) "" [1..5]`

f) `foldr const 'a' [1..5]`

g) `foldr const 0 "tacos"`

h) `foldl (flip const) 0 "burritos"`

i) `foldl (flip const) 'z' [1..5]`

Unconditional spine recursion

An important difference between `foldr` and `foldl` is that a left fold has the successive steps of the fold as its first argument. The next recursion of the spine isn't intermediated by the folding function as it is in `foldr`, which also means recursion of the spine is unconditional. Having a function that doesn't force evaluation of either of its arguments won't change anything. Let's review `const`:

```
Prelude> const 1 undefined
1
Prelude> (flip const) 1 undefined
*** Exception: Prelude.undefined
Prelude> (flip const) undefined 1
1
```

Now compare:

```
Prelude> xs = [1..5] ++ undefined
Prelude> foldr const 0 xs
1
Prelude> foldr (flip const) 0 xs
*** Exception: Prelude.undefined

Prelude> foldl const 0 xs
*** Exception: Prelude.undefined
Prelude> foldl (flip const) 0 xs
*** Exception: Prelude.undefined
```

However, while `foldl` unconditionally evaluates the spine, you can still selectively evaluate the values in the list. This will throw an error, because the bottom is part of the spine, and `foldl` must evaluate the spine:

```
Prelude> xs = [1..5] ++ undefined
```

```
Prelude> foldl (\_ _ -> 5) 0 xs
*** Exception: Prelude.undefined
```

But this is OK, because bottom is a value here:

```
Prelude> xs = [1..5] ++ [undefined]
Prelude> foldl (\_ _ -> 5) 0 xs
5
```

This feature means that `foldl` is generally inappropriate with lists that are or could be infinite, but the combination of the forced spine evaluation with non-strictness means that it is also usually inappropriate even for long lists, as the forced evaluation of the spine affects performance negatively. Because `foldl` must evaluate its whole spine before it starts evaluating values in each cell, it accumulates a pile of unevaluated values as it traverses the spine.

In most cases, when you need a left fold, you should use `foldl'`. This function, called "fold-l-prime," works the same way, except it is strict. In other words, it forces evaluation of the values inside the cons cells as it traverses the spine, rather than accumulating unevaluated expressions for each element of a list. The strict evaluation here means it has less negative effect on performance over long lists.

10.6 How to write fold functions

When we write folds, we begin by thinking about what our start value for the fold is. This is usually the identity value for the function. When we sum the elements of a list, the identity of summation is 0. When we multiply the elements of the list, the identity is 1. This start value is also our fallback in case the list is empty.

Next, we consider our arguments. A folding function takes two arguments, `a` and `b`, where `a` is always going to be one of the elements of the list, and `b` is either the start value or the value accumulated as the list is being processed.

Let's say we want to write a function to take the first three letters of each `String` value in a list of strings and concatenate that result into a final `String`. The type of the right fold for lists is:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

First, we'll set up the beginnings of our expression:

```
foldr (\a b -> undefined) []
  ["Pizza", "Apple", "Banana"]
```

We used an empty list as the start value, but since we plan to return a `String` as our result, we could be a little more explicit about our intent to build a `String` and make a small syntactic change:

```
foldr (\a b -> undefined) ""
  ["Pizza", "Apple", "Banana"]
```

Of course, because a `String` is a list, these are the same value:

```
Prelude> "" == []
True
```

But `""` signals intent with respect to the types involved:

```
Prelude> :t ""
"" :: [Char]
Prelude> :t []
[] :: [t]
```

Moving along, we next want to work on the function. We already know how to take the first three elements from a list, and we can reuse this for a `String`:

```
foldr (\a b -> take 3 a) ""
  ["Pizza", "Apple", "Banana"]
```

This will already type check and work, but it doesn't match the semantics we ask for:

```
Prelude> :{
*Main| let pab =
*Main|       ["Pizza", "Apple", "Banana"]
*Main| :}
Prelude> foldr (\a b -> take 3 a) "" pab
"Piz"
Prelude> foldl (\b a -> take 3 a) "" pab
"Ban"
```

We're only getting the first three letters of the first or the last string, depending on whether we do a right or left fold. Note the argument naming order, due to the difference in the types of `foldr` and `foldl`:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

```
foldl :: (b -> a -> b) -> b -> [a] -> b
```

The problem here is that right now, we're not folding the list. We're only mapping our `take 3` over the list and selecting the first or last result:

```
Prelude> map (take 3) pab
["Piz","App","Ban"]
Prelude> head $ map (take 3) pab
"Piz"
Prelude> last $ map (take 3) pab
"Ban"
```

So, let us make this a proper fold and accumulate the result by making use of the `b` argument. Remember, the `b` is the start value. Technically, we could use `concat` on the result of having mapped `take 3` over the list (or its reverse, if we want to simulate `foldl`):

```
Prelude> concat $ map (take 3) pab
"PizAppBan"
Prelude> rpab = reverse pab
Prelude> concat $ map (take 3) rpab
"BanAppPiz"
```

But we need an excuse to play with `foldr` and `foldl`, so we'll pretend none of this happened!

```
Prelude> f = (\a b -> take 3 a ++ b)
Prelude> foldr f "" pab
"PizAppBan"
Prelude> f' = (\b a -> take 3 a ++ b)
Prelude> foldl f' "" pab
"BanAppPiz"
```

Here, we are concatenating the result of having taken three elements from the string value in our input list onto the front of the string we're accumulating. If we want to be explicit, we can assert types for the values:

```
Prelude> :{
*Prelude| let f a b = take 3
*Prelude|             (a :: String) ++
*Prelude|             (b :: String)
*Prelude| :}
Prelude> foldr f "" pab
"PizAppBan"
```

If we assert something that isn't true, the type checker catches us:

```
Prelude> :{
*Prelude| let f a b = take 3 (a :: String)
*Prelude|             ++ (b :: [String])
*Prelude| :}

• Couldn't match type ‘[Char]’ with ‘Char’
  Expected type: [Char]
    Actual type: [String]
• In the second argument of ‘(++)’, namely
    ‘(b :: [String])’
  In the expression: take 3 (a :: String)
    ++ (b :: [String])
  In an equation for ‘f’:
      f a b = take 3 (a :: String)
        ++ (b :: [String])
```

This can be useful for checking that your mental model of the code is accurate.

Exercises: Database processing

Let's write some functions to process the following data:

```
import Data.Time

data DatabaseItem = DbString String
                  | DbNumber Integer
                  | DbDate   UTCTime
                  deriving (Eq, Ord, Show)

theDatabase :: [DatabaseItem]
theDatabase =
  [ DbDate (UTCTime
            (fromGregorian 1911 5 1)
    (secondsToDiffTime 34123))
  , DbNumber 9001
  , DbString "Hello, world!"
  , DbDate (UTCTime
            (fromGregorian 1921 5 1)
            (secondsToDiffTime 34123))
  ]
```

1. Write a function that filters for `DbDate` values and returns a list of the `UTCTime` values inside them:

   ```
   filterDbDate :: [DatabaseItem]
                -> [UTCTime]
   filterDbDate = undefined
   ```

2. Write a function that filters for `DbNumber` values and returns a list of the `Integer` values inside them:

   ```
   filterDbNumber :: [DatabaseItem]
                  -> [Integer]
   filterDbNumber = undefined
   ```

3. Write a function that gets the most recent date:

   ```
   mostRecent :: [DatabaseItem]
              -> UTCTime
   mostRecent = undefined
   ```

4. Write a function that sums all of the `DbNumber` values:

```
sumDb :: [DatabaseItem]
      -> Integer
sumDb = undefined
```

5. Write a function that gets the average of the `DbNumber` values:

```
-- You'll probably need to use fromIntegral
-- to get from Integer to Double.

avgDb :: [DatabaseItem]
      -> Double
avgDb = undefined
```

10.7 Folding and evaluation

What differentiates `foldr` and `foldl` is associativity. The right associativity of `foldr` means the folding function evaluates from the innermost cons cell to the outermost (the head). On the other hand, `foldl` recurses unconditionally to the end of the list through self-calls, and then the folding function evaluates from the outermost cons cell to the innermost:

```
Prelude> rcf = foldr (:) []
Prelude> xs = [1, 2, 3] ++ undefined
Prelude> take 3 $ rcf xs
[1,2,3]
Prelude> lcf = foldl (flip (:)) []
Prelude> take 3 $ lcf xs
*** Exception: Prelude.undefined
```

Let's dive into our `const` example a little more carefully:

```
foldr const 0 [1..5]
```

With `foldr`, you'll evaluate `const 1 (...)`, but `const` ignores the rest of the fold that would have occurred from the end of the list up to the number 1, so this returns 1 without having evaluated any more of the values or the spine. One way you could examine this for yourself would be:

```
Prelude> foldr const 0 ([1] ++ undefined)
1


Prelude> head ([1] ++ undefined)
1
Prelude> tail ([1] ++ undefined)
*** Exception: Prelude.undefined
```

Similarly for `foldl`:

```
foldl (flip const) 0 [1..5]
```

Here, `foldl` will recurse to the final cons cell, evaluate `(flip const) (...) 5`, ignore the rest of the fold that would occur from the beginning up to the number *5*, and return *5*.

The relationship between `foldr` and `foldl` is such that:

```
foldr f z xs =
  foldl (flip f) z (reverse xs)
```

But *only* for finite lists! Consider:

```
Prelude> xs = repeat 0 ++ [1,2,3]
Prelude> foldr const 0 xs
0
Prelude> xs' = repeat 1 ++ [1,2,3]
Prelude> rxs = reverse xs'
Prelude> foldl (flip const) 0 rxs
^CInterrupted.
-- ^^ bottom.
```

If we flip our folding function `f` and reverse the list `xs`, `foldr` and `foldl` will return the same result:

```
Prelude> xs = [1..5]
Prelude> foldr (:) [] xs
[1,2,3,4,5]
Prelude> foldl (flip (:)) [] xs
[5,4,3,2,1]
Prelude> foldl (flip (:)) [] (reverse xs)
[1,2,3,4,5]
```

```
Prelude> reverse $ foldl (flip (:)) [] xs
[1,2,3,4,5]
```

10.8 Summary

We presented a lot of material in this chapter. You might be feeling a little weary of folds right now. So what's the executive summary?

foldr

1. The rest of the fold (recursive invocation of `foldr`) is an argument to the folding function you pass to `foldr`. It doesn't directly self-call as a tail-call like `foldl`. You could think of it as alternating between applications of `foldr` and your folding function `f`. The next invocation of `foldr` is conditional on `f` having asked for more of the results of having folded the list. That is:

   ```
   foldr :: (a -> b -> b) -> b -> [a] -> b
   --              ^
   ```

   That `b` we're pointing at in `(a -> b -> b)` is *the rest of the fold*. Evaluating that evaluates the next application of `foldr`.

2. Associates to the right.

3. Works with infinite lists. We know this because:

   ```
   Prelude> foldr const 0 [1..]
   1
   ```

4. Is a good default choice whenever you want to transform data structures, be they finite or infinite.

foldl

1. Self-calls (using tail calls) through the list, only beginning to produce values after reaching the end of the list.

2. Associates to the left.

3. Cannot be used with infinite lists. Try the infinite list example earlier, and your REPL will hang.

4. Is nearly useless and should almost always be replaced with `foldl'` for reasons we'll explain later when we talk about writing efficient Haskell programs.

10.9 Scans

Scans, which we have mentioned above, work similarly to maps and also to folds. Like folds, they accumulate values instead of keeping a list's individual values separate. Like maps, they return a list of results. In this case, the list of results shows the intermediate stages of evaluation, that is, the values that accumulate as the function is doing its work.

Scans are not used as frequently as folds, and once you understand the basic mechanics of folding, there isn't a whole lot new to understand. Still, it is useful to know about them and get an idea of why you might need them.[3]

First, let's take a look at the types. We'll do a direct comparison of the types of folds and scans, so the differences are clear:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
scanr :: (a -> b -> b) -> b -> [a] -> [b]

foldl :: (b -> a -> b) -> b -> [a] -> b
scanl :: (b -> a -> b) -> b -> [a] -> [b]
```

The primary difference is that the final result is a list (a fold *can* return a list as a result, as well, but they don't always). This means that they are not catamorphisms and, in an important sense, aren't folds at all. But no matter! The type signatures are similar, and the routes of spine traversal and evaluation are similar. This does mean that you can use scans in places where you can't use a fold, precisely because you return a list of results rather than reducing the spine of the list.

The results that scans produce can be represented like this:

```
scanr (+) 0 [1..3]
```

[3]The truth is that scans are not used often, but there are times when you want to fold a function over a list and return a list of the intermediate values that you can then use as input to some other function. For a particularly elegant use case, please see Chris Done's blog post: `http://chrisdone.com/posts/twitter-problem-loeb`.

```
[1 + (2 + (3 + 0)), 2 + (3 + 0), 3 + 0, 0]
[6, 5, 3, 0]

scanl (+) 0 [1..3]
[0, 0 + 1,0 + 1 + 2, 0 + 1 + 2 + 3]
[0, 1, 3, 6]
```

```
scanl (+) 1 [1..3]

-- unfolding the
-- definition of scanl
= [ 1, 1 + 1
  , (1 + 1) + 2
  , ((1 + 1) + 2) + 3
  ]

-- evaluating addition
= [1, 2, 4, 7]
```

Then, to make this more explicit and properly equational, we can follow along with how `scanl` expands for this expression based on the definition. First, we must see how `scanl` is defined. We're going to show you a version of it from a slightly older `base` library for GHC Haskell. The differences don't change anything important for us here:

```
scanl :: (a -> b -> a) -> a -> [b] -> [a]
scanl f q ls =
  q : (case ls of
         []   -> []
         x:xs -> scanl f (f q x) xs)
```

In an earlier chapter, we wrote a recursive function that returns the nth Fibonacci number. You can use a scan function to return a list of Fibonacci numbers. We're going to do this in a source file, because it will, in this state, return an infinite list (feel free to try loading it into your REPL and running it, but be quick with the Ctrl-C):

```
fibs = 1 : scanl (+) 1 fibs
```

We start with a value of 1 and cons that onto the front of the list generated by our scan. The list itself has to be recursive, because, as we saw previously, the idea of Fibonacci numbers is that each one is the sum of the previous two in the sequence; scanning the results of `+` over a non-recursive list of numbers whose start value is 1 would give us this:

```
scanl (+) 1 [1..3]
[1, 1 + 1, (1 + 1) + 2, ((1 + 1) + 2) + 3]
[1,2,4,7]
```

Instead of the `[1, 1, 2, 3, 5, ...]` that we're looking for.

Getting the Fibonacci number we want

But we don't really want an infinite list of Fibonacci numbers; that isn't very useful. We need a method to either take some number of elements from that list or find the nth element as we did before. Fortunately, that's the easy part. We'll use the "bang bang" operator, `!!`, to find the nth element. This operator is a way to index into a list, and indexing in Haskell starts from 0. That is, the first value in your list is indexed as 0. But, otherwise, the operator is straightforward:

```
(!!) :: [a] -> Int -> a
```

It needs a list as its first argument, an `Int` as its second argument, and it returns one element from the list. Which item it returns is the value that is in the nth spot, where n is our `Int`. Let's modify our source file:

```
fibs    = 1 : scanl (+) 1 fibs
fibsN x = fibs !! x
```

Once we load the file into our REPL, we can use `fibsN` to return the nth element of our scan:

```
Prelude> fibsN 0
1
Prelude> fibsN 2
2
Prelude> fibsN 6
13
```

Now, you can modify your source code to use the `take` or `takeWhile` functions or to filter it in any way you like. One note: filtering without also taking won't work too well, because you're still getting an infinite list. It's a filtered infinite list, sure, but still infinite.

Scans exercises

1. Modify your `fibs` function to only return the first 20 Fibonacci numbers.
2. Modify `fibs` to return the Fibonacci numbers that are less than 100.
3. Try to write the `factorial` function from Chapter 8 as a scan. You'll want `scanl` again, and your start value will be 1. Warning: this will also generate an infinite list, so you may want to pass it through a `take` function or similar.

10.10 Chapter exercises

Warm-up and review

For the following set of exercises, you are not expected to use folds. These are intended to review material from previous chapters. Feel free to use any syntax or structure from previous chapters that seems appropriate.

1. Given the following sets of consonants and vowels:

```
stops  = "pbtdkg"
vowels = "aeiou"
```

   a) Write a function that takes inputs from `stops` and `vowels` and makes 3-tuples of all possible stop-vowel-stop combinations. These will not all correspond to real words in English, although the stop-vowel-stop pattern is common enough that many of them will.

   b) Modify that function so that it only returns the combinations that begin with a `p`.

c) Now set up lists of nouns and verbs (instead of stops and vowels), and modify the function to make tuples representing possible noun-verb-noun sentences.

2. What does the following mystery function do? What is its type? Try to get a good sense of what it does before you test it in the REPL to verify it:

```
seekritFunc x =
  div (sum (map length (words x)))
      (length (words x))
```

3. We'd really like the answer to be more precise. Can you rewrite that using fractional division?

Rewriting functions using folds

In the previous chapter, you wrote these functions using direct recursion over lists. The goal now is to rewrite them using folds. Where possible, to gain a deeper understanding of folding, try rewriting the fold version so that it is point-free.

Point-free versions of these functions written with a fold should look like this:

```
myFunc = foldr f z
```

So, for example, with the `and` function:

```
-- direct recursion, not using &&
myAnd :: [Bool] -> Bool
myAnd [] = True
myAnd (x:xs) =
  if x == False
  then False
  else myAnd xs

-- direct recursion, using &&
myAnd :: [Bool] -> Bool
myAnd [] = True
myAnd (x:xs) = x && myAnd xs
```

```
-- fold, not point-free
myAnd :: [Bool] -> Bool
myAnd = foldr
        (\a b ->
          if a == False
          then False
          else b) True

-- fold, both myAnd and the folding
-- function are point-free now
myAnd :: [Bool] -> Bool
myAnd = foldr (&&) True
```

The goal here is to converge on the final version where possible. You don't need to write all variations for each example, but the more variations you write, the deeper your understanding of these functions will become.

1. `myOr` returns `True` if any `Bool` in the list is `True`:

   ```
   myOr :: [Bool] -> Bool
   myOr = undefined
   ```

2. `myAny` returns `True` if `a -> Bool` applied to any of the values in the list returns `True`:

   ```
   myAny :: (a -> Bool) -> [a] -> Bool
   myAny = undefined
   ```

   Example for validating `myAny`:

   ```
   Prelude> myAny even [1, 3, 5]
   False
   Prelude> myAny odd [1, 3, 5]
   True
   ```

3. Write two versions of `myElem`. One version should use folding and the other should use `any`:

   ```
   myElem :: Eq a => a -> [a] -> Bool
   ```

```
Prelude> myElem 1 [1..10]
True
Prelude> myElem 1 [2..10]
False
```

4. Implement `myReverse`. Don't worry about trying to make it lazy:

```
myReverse :: [a] -> [a]
myReverse = undefined

Prelude> myReverse "blah"
"halb"
Prelude> myReverse [1..5]
[5,4,3,2,1]
```

5. Write `myMap` in terms of `foldr`. It should have the same behavior as the built-in `map`:

```
myMap :: (a -> b) -> [a] -> [b]
myMap = undefined
```

6. Write `myFilter` in terms of `foldr`. It should have the same behavior as the built-in `filter`:

```
myFilter :: (a -> Bool) -> [a] -> [a]
myFilter = undefined
```

7. `squish` flattens a list of lists into a list:

```
squish :: [[a]] -> [a]
squish = undefined
```

8. `squishMap` maps a function over a list and concatenates the result:

```
squishMap :: (a -> [b]) -> [a] -> [b]
squishMap = undefined

Prelude> squishMap (\x -> [1, x, 3]) [2]
[1,2,3]
Prelude> f x = "WO " ++ [x] ++ " OT "
Prelude> squishMap f "blah"
"WO b OT WO l OT WO a OT WO h OT "
```

9. `squishAgain` flattens a list of lists into a list. This time, re-use the `squishMap` function:

```
squishAgain :: [[a]] -> [a]
squishAgain = undefined
```

10. `myMaximumBy` takes a comparison function and a list and returns the greatest element of the list based on the last value that the comparison returns `GT` for:

```
myMaximumBy :: (a -> a -> Ordering)
            -> [a]
            -> a
myMaximumBy = undefined

Prelude> myMaximumBy (\_ _ -> GT) [1..10]
1
Prelude> myMaximumBy (\_ _ -> LT) [1..10]
10
Prelude> myMaximumBy compare [1..10]
10
```

11. `myMinimumBy` takes a comparison function and a list and returns the least element of the list based on the last value that the comparison returns `LT` for:

```
myMinimumBy :: (a -> a -> Ordering)
            -> [a]
            -> a
myMinimumBy = undefined

Prelude> myMinimumBy (\_ _ -> GT) [1..10]
10
Prelude> myMinimumBy (\_ _ -> LT) [1..10]
1
Prelude> myMinimumBy compare [1..10]
1
```

10.11 Definitions

1. A *fold* is a higher-order function which, given a function to accumulate the results and a recursive data structure, returns the built up value. Usually a "start value" for the accumulation is provided along with a function that can combine the type of values in the data structure with the accumulation. The term fold is typically used with reference to collections of values referenced by a recursive datatype. For a generalization of "breaking down structure," see *catamorphism*.

2. A *catamorphism* is a generalization of folds to arbitrary datatypes. Where a fold allows you to break down a list into an arbitrary datatype, a catamorphism is a means of breaking down the structure of any datatype. The `bool :: a -> a -> Bool -> a` function in `Data.Bool` is an example of a simple catamorphism for a simple, non-collection datatype. Similarly, `maybe :: b -> (a -> b) -> Maybe a -> b` is the catamorphism for `Maybe`. See if you can notice a pattern:

```
data Bool = False | True
bool :: a -> a -> Bool -> a

data Maybe a = Nothing | Just a
maybe :: b -> (a -> b) -> Maybe a -> b

data Either a b = Left a | Right b
either :: (a -> c)
       -> (b -> c)
       -> Either a b
       -> c
```

3. A *tail call* is the final result of a function. Some examples of tail calls in Haskell functions:

```
f x y z = h (subFunction x y z)
   where subFunction x y z = g x y z
-- the "tail call" is
-- h (subFunction x y z)
-- or, more precisely, h
```

4. *Tail recursion* occurs in a function whose tail calls are recursive invocations of itself. This is distinguished from functions that call other functions in their tail call. For example:

```
f x y z = h (subFunction x y z)
   where subFunction x y z = g x y z
```

The above is not tail recursive, since it calls `h`, not itself.

```
f x y z = h (f (x - 1) y z)
```

Still not tail recursive. `f` is invoked again but not in the tail call of `f`. It's an argument to the tail call, `h`:

```
f x y z = f (x - 1) y z
```

This is tail recursive. `f` is calling itself directly with no intermediaries.

```
foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)
```

Not tail recursive—we give up control to the combining function `f` before continuing through the list. `foldr`'s recursive calls will bounce between `foldr` and `f`.

```
foldl f z []     = z
foldl f z (x:xs) = foldl f (f z x) xs
```

Tail recursive. `foldl` invokes itself recursively. The combining function is only an argument to the recursive fold.

10.12 Follow-up resources

1. Antoni Diller. *Introduction to Haskell*. Unit 6.
   `http://www.cantab.net/users/antoni.diller/haskell/haskell.html`
2. Graham Hutton. *A tutorial on the universality and expressiveness of fold*.
   `http://www.cs.nott.ac.uk/~gmh/fold.pdf`

Chapter 11

Algebraic Datatypes

> The most depressing thing about life as a programmer, I think, is if you're faced with a chunk of code that either someone else wrote or, worse still, you wrote yourself but no longer dare to modify. That's depressing.
>
> Simon Peyton Jones

11.1 Algebraic datatypes

We have spent a lot of time talking about datatypes already, so you may think we've covered everything that needs to be said about those. This chapter's purpose is ultimately to explain how to construct your own datatypes in Haskell. Writing your own datatypes can help you leverage some of Haskell's most powerful features—pattern matching, type checking, and inference—in a way that makes your code more concise and safer. But to understand that, first we need to explain the differences among datatypes more fully and understand what it means when we say datatypes are *algebraic*.

A type can be thought of as an enumeration of constructors that have zero or more arguments.[1] We will return to this description throughout the chapter, each time emphasizing a different portion of it.

Haskell offers sum types, product types, product types with record syntax, type aliases (for example, `String` is a type alias for `[Char]`), and a special datatype called a *newtype* that provides for a different set of options and constraints from either type synonyms or data declarations. We will explain each of these in detail in this chapter and show you how to exploit them for maximum utility and type safety.

This chapter will:

- Explain the "algebra" of algebraic datatypes.
- Analyze the construction of data constructors.
- Spell out when and how to write your own datatypes.
- Clarify the usage of type synonyms and *newtypes*.
- Introduce *kinds*.

11.2 Data declarations review

We often want to create custom datatypes for structuring and describing the data we are processing. Doing so can help you analyze your problem by allowing you to focus first on how you *model* the

[1] This description, slightly edited for our purposes, was proposed by Orah Kittrell in the `#haskell-beginners` IRC channel.

domain before you begin thinking about how you write *computations* that solve your problem. It can also make your code easier to read and use, because it lays the domain model out clearly.

In order to write your own types, though, you must understand the way datatypes are constructed in more detail than we've covered so far. Let's begin with a review of the important parts of datatypes, using the data declarations for `Bool` and lists:

```
  data Bool  =  False  |  True
-- [1]  [2] [3]  [4]  [5] [6]

data [] a = [ ]   | a  :  [a]
--   [ 7 ]  [8]       [9]
```

1. Keyword `data` to signal that what follows is a data declaration, or a declaration of a datatype.
2. Type constructor (with no arguments).
3. The equals sign divides the type constructor from its data constructors.
4. Data constructor. In this case, a data constructor that takes no arguments, which is called a *nullary* constructor. This is one of the possible values of this type that can show up in term-level code.
5. The pipe denotes a sum type, which indicates a logical disjunction (colloquially, *or*) in which values can have this type.
6. Constructor for the value `True`, another nullary constructor.
7. Type constructor with an argument. An empty list has to be applied to an argument in order to become a list of *something*. Here the argument is a polymorphic type variable, so the list's argument can be of different types.
8. Data constructor for the empty list.
9. Data constructor that takes two arguments: an `a` and also an `[a]`.

When we talk about a data declaration, we are talking about the definition of the entire type. If we think of a type as "an enumeration of constructors that have *zero* or more arguments," then `Bool` is an enumeration of two possible constructors, each of which takes *zero* arguments, while the type constructor `[]` enumerates two possible constructors and one of them takes *two* arguments. The pipe denotes what we call a *sum type*, a type that has *more than one* constructor inhabiting it.

In addition to sum types, Haskell also has *product types*, and we'll talk more about those in a bit. The data constructors in product types have more than one parameter. But first, let's turn our attention to the meaning of the word *constructor*.

11.3 Data and type constructors

There are two kinds of constructors in Haskell: type constructors and data constructors. Type constructors are used only at the type level, in type signatures and type class declarations and instances. Types are static and resolve at compile time. Data constructors construct the values at term level, values you can interact with at runtime. We call them constructors, because they define a means of creating or building a type or a value.

Although the term *constructor* is often used to describe all type constructors and data constructors, we can make a distinction between *constants* and *constructors*. Type and data constructors that take no arguments are constants. They can only store a fixed type and amount of data. So, in the `Bool` datatype, for example, `Bool` is a type constant, a concrete type that isn't waiting for any additional information in the form of an argument in order to be fully realized as a type. It enumerates two values that are also constants, `True` and `False`, because they take no arguments. While we call `True` and `False` "data constructors," in fact since they take no arguments, their values are already established and not being *constructed* in any meaningful sense.

However, sometimes we need the flexibility of allowing different types or amounts of data to be stored in our datatypes. For those times, type and data constructors may be parameterized. When a constructor takes an argument, then it's like a function in at least one

sense—it must be *applied* to become a concrete type or value. The following datatypes are pseudonymous versions of real datatypes in Haskell. We've given them pseudonyms, because we want to focus on the syntax, not the semantics, for now:

```
data Trivial = Trivial'
--   [1]      [2]

data UnaryTypeCon a = UnaryValueCon a
--       [3]               [4]
```

1. Here, the type constructor `Trivial` is like a constant value but at the type level. It takes no arguments and is thus *nullary*. The Haskell Report calls these *type constants* to distinguish them from type constructors that take arguments.
2. The data constructor `Trivial'` is also like a constant value, but it exists in value, term, or runtime space. These are not three different things, but three different words for the same space that types serve to describe.
3. `UnaryTypeCon` is a type constructor of one argument. It's a constructor awaiting a type constant to be applied to, but it has no behavior in the sense that we think of functions as having. Such type-level functions exist but are not covered in this book.
4. `UnaryValueCon` is a data constructor of one argument awaiting a value to be applied to. Again, it doesn't behave like a term-level function in the sense of performing an operation on data. It's more like a box to put values into. Be careful with the box/container analogy, as it will betray you later—not all type arguments to constructors have value-level witnesses! Some are *phantom*.

Each of these datatypes only enumerates one data constructor. Whereas `Trivial'` is the only possible concrete value for type `Trivial`, `UnaryValueCon` could show up as different literal values at runtime, depending on what type of `a` it is applied to. Think back to the list datatype: at the type level, you have `a : [a]`, where the `a` is a variable. At the term level, in your code, that will be applied to some type

of values and become, for example, `[Char]` or `[Integer]` (or a list of whatever other concrete type—obviously, the set of possible lists is pretty big).

11.4 Type constructors and kinds

Let's look again at the list datatype:

```
data [] a = []  | a  :  [a]
```

This must be applied to a concrete type before you have a list. We can see the parallel with functions when we look at the *kind* signature.

Kinds are the types of types, or types one level up. We represent kinds in Haskell with `*`. We know something is a fully applied, concrete type when it is represented as `*`. When it is `* -> *`, it, like a function, is still waiting to be applied.

Compare the following:

```
Prelude> f = not True
Prelude> :t f
f :: Bool

Prelude> f x = x > 3
Prelude> :t f
f :: (Ord a, Num a) => a -> Bool
```

The first `f` takes no arguments and is not awaiting application to anything in order to produce a value, so its type signature is a concrete type—note the lack of a function arrow. But the second `f` is awaiting application to an `x`, so its type signature has a function arrow. Once we apply it to a value, it also has a concrete type:

```
Prelude> f x = x > 3
Prelude> :t f 5
f 5 :: Bool
```

We query the kind signature of a type constructor (not a data constructor) in GHCi with a `:kind` or `:k`. We see that kind signatures give us similar information about type constructors:

```
Prelude> :k Bool
Bool :: *
Prelude> :k [Int]
[Int] :: *
Prelude> :k []
[] :: * -> *
```

Both `Bool` and `[Int]` are fully applied, concrete types, so their kind signatures have no function arrows. That is, they are not awaiting application to anything in order to be fully realized. The kind of `[]`, though, is `* -> *`, because it still needs to be applied to a concrete type before it becomes itself a concrete type. This is what the *constructor* of "type constructor" is referring to.

11.5 Data constructors and values

We mentioned a bit ago that the Haskell Report draws a distinction between type *constants* and type *constructors*. We can draw a similar distinction between data constructors and constant values:

```
data PugType = PugData
--   [1]        [2]

data HuskyType a = HuskyData
--      [3]          [4]

data DogueDeBordeaux doge =
--       [5]
    DogueDeBordeaux doge
--       [6]
```

1. `PugType` is the type constructor, but it takes no arguments, so we can think of it as being a type *constant*. This is how the Haskell Report refers to such types. This type enumerates one constructor.

2. `PugData` is the only data constructor for the type `PugType`. It also happens to be a *constant value*, because it takes no arguments and stands only for itself. For any function that requires a value of type `PugType`, you know that value will be `PugData`.

3. `HuskyType` is the type constructor, and it takes a single parametrically polymorphic type variable as an argument. It also enumerates one data constructor.

4. `HuskyData` is the data constructor for `HuskyType`. Note that the type variable argument `a` does *not* occur as an argument to `HuskyData` or anywhere else after the `=`. That means our type argument `a` is a *phantom*, or, "has no witness." We will elaborate on this later. Here, `HuskyData` is a constant value, like `PugData`.

5. `DogueDeBordeaux` is a type constructor and has a single type variable argument like `HuskyType`, but called `doge` instead of `a`. Why? Because the names of variables don't matter. At any rate, this type also enumerates one constructor.

6. `DogueDeBordeaux` is the lone data constructor. It has the same name as the type constructor, but they are not the same thing. The `doge` type variable in the type constructor occurs also in the data constructor. Remember that, because they are the same type variable, these must agree with each other: `doge` must equal `doge`. If your type is `DogueDeBordeaux [Person]`, you must necessarily have a list of `Person` values contained in the `DogueDeBordeaux` value. But because `DogueDeBordeaux` must be applied before it's a concrete value, its literal value at runtime can change:

```
Prelude> :t DogueDeBordeaux
DogueDeBordeaux :: doge
                -> DogueDeBordeaux doge
```

   We can query the type of the value (not the type constructor but the data constructor—it can be confusing when the type constructor and the data constructor have the same name, but it's pretty common to do that in Haskell, because the compiler doesn't confuse type names with value names the way we mortals do). It tells us that once `doge` is bound to a concrete type, then this will be a value of type `DogueDeBordeaux doge`. It isn't a value yet, but it's a definition for how to construct a value of that type.

Here's how to make a value of the type of each:

```
myPug = PugData :: PugType

myHusky :: HuskyType a
myHusky = HuskyData

myOtherHusky :: Num a => HuskyType a
myOtherHusky = HuskyData


myOtherOtherHusky :: HuskyType [[[[Int]]]]
myOtherOtherHusky = HuskyData
-- no witness to the contrary       ^
```

This will work, because the value 10 agrees with the type variable being bound to `Int`:

```
myDoge :: DogueDeBordeaux Int
myDoge = DogueDeBordeaux 10
```

This will not work, because 10 cannot be reconciled with the type variable being bound to `String`:

```
badDoge :: DogueDeBordeaux String
badDoge = DogueDeBordeaux 10
```

Given this, we can see that constructors are how we create values of types and refer to types in type signatures. There's a parallel here between type constructors and data constructors that should be noted. We can illustrate this with a new, canine-oriented datatype:

```
data Doggies a =
    Husky a
  | Mastiff a
  deriving (Eq, Show)

-- type constructor awaiting an argument
Doggies
```

Note that the kind signature for the type constructor looks like a function, and the type signature for either of its data constructors looks similar.

This needs to be applied to become a concrete type:

```
Prelude> :k Doggies
Doggies :: * -> *
```

And this needs to be applied to become a concrete value:

```
Prelude> :t Husky
Husky :: a -> Doggies a
```

So, the behavior of constructors is such that if they don't take any arguments, they behave like (type or value-level) constants. If they do take arguments, they act like (type or value-level) functions that don't *do* anything except get applied.

Exercises: Dog types

Given the datatypes defined in the above sections:

1. Is `Doggies` a type constructor or a data constructor?
2. What is the kind of `Doggies`?
3. What is the kind of `Doggies String`?
4. What is the type of `Husky 10`?
5. What is the type of `Husky (10 :: Integer)`?
6. What is the type of `Mastiff "Scooby Doo"`?
7. Is `DogueDeBordeaux` a type constructor or a data constructor?
8. What is the type of `DogueDeBordeaux`?
9. What is the type of `DogueDeBordeaux "doggie!"`

11.6 What's a type and what's data?

As we said, types are static and resolve at compile time. Types are known before runtime, whether through explicit declaration or type inference, and that's what makes them static types. Information about types, however, does not persist through to runtime. Data are what we're working with at runtime.

Compile time is literally when your program is getting compiled by GHC or checked before execution in a REPL like GHCi. Runtime is the actual execution of your program. Types circumscribe values, and in that way, they describe which values are flowing through which parts of your program:

```
type constructors --   compile-time

--------------------   phase separation

data constructors --   runtime
```

Both data constructors and type constructors begin with capital letters, but a constructor *before* the `=` in a datatype definition is a type constructor, while constructors *after* the `=` are data constructors. Data constructors are usually generated by the declaration. One tricky bit here is that when data constructors take arguments, those arguments refer to *other types*. Because of this, not everything referred to in a datatype declaration is necessarily *generated* by that datatype itself. Let's take a look at a short example with different datatypes to demonstrate what we mean by this.

We start with a datatype `Price` that has one type constructor, one data constructor, and one type argument in the data constructor:

```
data Price =
--      (a)
  Price Integer deriving (Eq, Show)
-- (b)    [1]
```

The type constructor is a. The data constructor is b, and it takes one type argument, 1.

The value `Price` does not depend solely on this datatype definition. It depends on the type `Integer`, as well. If, for some reason, `Integer` isn't in scope, we'd be unable to generate `Price` values.

Next, we'll define two datatypes, `Manufacturer` and `Airline`, that are each sum types with three data constructors. Each data constructor in these is a possible value of that type, and since none of them take arguments, all are generated by their declarations and are more like constant values than constructors:

```
data Manufacturer =
--       (c)
              Mini
--            (d)
            | Mazda
--            (e)
            | Tata
--            (f)
              deriving (Eq, Show)
```

Manufacturer has the type constructor c. `Manufacturer` has three data constructors: d, e, and f

```
data Airline =
--     (g)
         PapuAir
--         (h)
       | CatapultsR'Us
--         (i)
       | TakeYourChancesUnited
--         (j)
         deriving (Eq, Show)
```

The type constructor is g. `Airline` has three data constructors: h, i, and j.

Next, we'll look at another sum type, but this one has data constructors that take arguments. For the type `Vehicle`, the data constructors are `Car` and `Plane`, so a `Vehicle` is either a `Car` value or a `Plane` value. They each take types as arguments, just as `Price` itself took the type `Integer` as an argument:

```
data Vehicle = Car Manufacturer Price
--    (k)      (l)     [2]       [3]
             | Plane Airline
--             (m)    [4]
             deriving (Eq, Show)
```

The type constructor is k. There are two data constructors, l and m. The type arguments are numbered 2, 3, and 4. 2 and 3 are type arguments to the data constructor `Car`, while 4 is the type argument to the data constructor `Plane`. To construct a `Plane` value, therefore, we need a value from the `Airline` type.

In the above, the datatypes generate the constructors marked with a letter. The type arguments marked with a number existed prior to the declarations. Their definitions exist outside of this declaration, and they must be in scope to be used as part of this declaration.

Each of the above datatypes has a `deriving` clause. We have seen this before, as you will usually want to derive an instance of `Show` for the datatypes you write. The instance allows your data to be printed to the screen as a string. Deriving `Eq` is also common and allows you to derive equality operations automatically for most datatypes where that would make sense. There are other type classes that allow derivation in this manner, and it obviates the need for manually writing instances for each datatype and type class (reminder: you saw an example of this in chapter 6, on type classes).

As we've seen, data constructors can take arguments. Those arguments will be specific types but not specific values. In standard Haskell, we can't choose specific values of types as type arguments. We can't say, for example, "`Bool` without the possibility of `False` as a value." If you accept `Bool` as a valid type for a function or as the component of a datatype, you must accept all of `Bool`.

Exercises: Vehicles

For these exercises, we'll use the datatypes defined in the above section. It would be good if you've typed them all into a source file already, but if you haven't, please do so now. You can then define some sample data on your own, or use these to get you started:

```
myCar    = Car Mini (Price 14000)
urCar    = Car Mazda (Price 20000)
clownCar = Car Tata (Price 7000)
doge     = Plane PapuAir
```

1. What is the type of `myCar`?
2. Given the following, define the functions:

   ```
   isCar :: Vehicle -> Bool
   isCar = undefined

   isPlane :: Vehicle -> Bool
   isPlane = undefined

   areCars :: [Vehicle] -> [Bool]
   areCars = undefined
   ```

3. Now, we're going to write a function to tell us the manufacturer of a piece of data:

   ```
   getManu :: Vehicle -> Manufacturer
   getManu = undefined
   ```

4. Given that we're returning the `Manufacturer`, what will happen if you use this on `Plane` data?
5. All right. Let's say you decide to add the size of the plane as an argument to the `Plane` constructor. Add that to your datatypes in the appropriate places, and change your data and functions appropriately.

11.7 Data constructor arities

Now that we have a good understanding of the anatomy of datatypes, we want to start demonstrating why we call them "algebraic." We'll start by looking at something called *arity*. Arity refers to the number of arguments a function or constructor takes. A function that takes no arguments is called *nullary*, where nullary is a contraction of "null" and "-ary." Null means zero, the "-ary" suffix means "of or pertaining

to." "-ary" is a common suffix used when talking about mathematical arity, such as with nullary, unary, binary, and the like.

Data constructors that take no arguments are also called nullary. Nullary data constructors, such as `True` and `False`, are constant values at the term level and, since they have no arguments, they can't construct or represent any data other than themselves. They are values that stand for themselves and act as witnesses of the datatype in which they were declared.

We've said that "a type can be thought of as an enumeration of constructors that have zero or *more* arguments." We'll look next at constructors with arguments.

We've seen how data constructors may take arguments and that makes them more like functions, in that they must be applied to something before you have a value. Data constructors that take one argument are called *unary*. As we will see later in this chapter, data constructors that take more than one argument are called *products*.

All of the following are valid data declarations:

```
-- nullary
data Example0 =
  Example0
  deriving (Eq, Show)

-- unary
data Example1 =
  Example1 Int
  deriving (Eq, Show)

-- product of Int and String
data Example2 =
  Example2 Int String
  deriving (Eq, Show)
```

```
Prelude> Example0
Example0
Prelude> Example1 10
Example1 10
Prelude> Example1 10 == Example1 42
```

```
False
Prelude> nc = Example2 1 "NC"
Prelude> Example2 10 "FlappityBat" == nc
False
```

Our `Example2` is an example of a *product*, which is like a tuple and which we've seen before. Tuples can take several arguments—as many as there are inhabitants of the tuple—and are considered the canonical product type, but they are *anonymous products*, because they have no name. We'll talk more about product types soon.

Unary (one argument) data constructors contain a single value of whatever type their argument has. The following is a data declaration that contains the data constructor `MyVal`. `MyVal` takes one `Int` argument and creates a type named `MyType`:

```
data MyType = MyVal Int
--   [1]      [2]   [3]
     deriving (Eq, Show)
--     [4]      [5]
```

1. Type constructor.
2. Data constructor. `MyVal` takes one type argument, so it is called a *unary* data constructor.
3. Type argument to the data constructor.
4. Deriving clause.
5. Type class instances being derived. We're getting equality `Eq` and value stringification `Show` for free.

```
Prelude> :t MyVal
MyVal :: Int -> MyType
Prelude> MyVal 10
MyVal 10
Prelude> MyVal 10 == MyVal 10
True
Prelude> MyVal 10 == MyVal 9
False
```

Because `MyVal` has one `Int` argument, a value of type `MyType` must contain one—and only one—`Int` value.

11.8 What makes these datatypes algebraic?

Algebraic datatypes in Haskell are algebraic, because we can describe the patterns of argument structures using two basic operations: sum and product. The most direct way to explain why they're called sum and product is to demonstrate sum and product in terms of *cardinality*. This can be understood in terms of the cardinality you see with finite sets.[2] This doesn't map perfectly, as we can have infinite data structures in Haskell, but it's a good way to begin understanding and appreciating how datatypes work. When it comes to programming languages, we are concerned with *computable* functions, not just those that can generate a set.

The cardinality of a datatype is the number of possible values it defines. That number can be as small as 0 or as large as infinity (for example, numeric datatypes and lists). Knowing how many possible values inhabit a type can help you reason about your programs. In the following sections, we'll show you how to calculate the cardinality of a given datatype based solely on how it is defined. From there, we can determine how many different *possible* implementations there are of a function for a given type signature.

Before we get into the specifics of how to calculate cardinality in general, we're going to take cursory glances at some datatypes with easy to understand cardinalities: `Bool` and `Int`.

We've looked extensively at the `Bool` type already, so you already know it only has two inhabitants that are both nullary data constructors, so `Bool` only has two possible values. The cardinality of `Bool` is, therefore, 2. Even without understanding the rules of cardinality of sum types, we can see why this is true.

Another set of datatypes with cardinality that is reasonably easy to understand are the `Int` types. In part, this is because `Int` and the related types `Int8`, `Int16`, and `Int32` have clearly delineated upper and lower bounds, defined by the amount of memory they are permitted to use. We'll use `Int8` here, even though it isn't very common in Haskell, because it has the smallest set of possible inhabitants, and

[2]Type theory was developed as an alternative mathematical foundation to set theory. We won't write formal proofs based on this, but the way we reason informally about types as programmers derives in part from their origins as sets. Finite sets contain a number of unique objects; that number is called their cardinality.

thus the arithmetic is a bit easier to do. Valid `Int8` values are whole numbers from -128 to 127.

`Int8` is not included in the standard `Prelude`, unlike standard `Int`, so we need to import it to see it in the REPL, but after we do that we can use `maxBound` and `minBound` from the `Bounded` type class to view their upper and lower values:

```
Prelude> import Data.Int

Prelude Data.Int> minBound :: Int8
-128
Prelude Data.Int> maxBound :: Int8
127
```

Given that this range includes the value 0, we can easily figure out the cardinality of `Int8` with some quick addition: 128 + 127 + 1 = 256. So the cardinality of `Int8` is 256. Anywhere in your code where you'd have a value of type `Int8`, there are 256 possible runtime values.

Exercises: Cardinality

While we haven't explicitly described the rules for calculating the cardinality of datatypes yet, you might already have an idea of how to do it for simple datatypes with nullary constructors. Try not to overthink these exercises—follow your intuition based on what you know:

1. ```
   data PugType = PugData
   ```
2. For this one, recall that `Bool` is also defined with the `|` symbol:

   ```
   data Airline =
         PapuAir
       | CatapultsR'Us
       | TakeYourChancesUnited
   ```
3. Given what we know about `Int8`, what's the cardinality of `Int16`?
4. Use the REPL and `maxBound` and `minBound` to examine `Int` and `Integer`. What can you say about the cardinality of those types?
5. Extra credit (impress your friends!): what's the connection between the 8 in `Int8` and that type's cardinality of 256?
   ```

Simple datatypes with nullary data constructors

We'll start our exploration of cardinality by looking at datatypes with nullary data constructors:

```
data Example = MakeExample deriving Show
```

`Example` is our type constructor, and `MakeExample` is our only data constructor. Since `MakeExample` takes no type arguments, it is a nullary constructor. We know that nullary data constructors are constants and represent only themselves as values. It is a single value whose only content is its name, not any other data. We can think of nullary constructors as representing *one* value, when we reason about the cardinality of the types they inhabit.

All you can say about `MakeExample` is that the constructor is the value `MakeExample` and that it inhabits the type `Example`.

There, the only inhabitant is `MakeExample`. Given that `MakeExample` is a single nullary value, the cardinality of the type `Example` is 1. This is useful, because it tells us that any time we see `Example` in the type signature of a function, we only have to reason about one possible value.

Exercises: For example

1. You can query the type of a value in GHCi with the `:type` command, also abbreviated `:t`.

   Example:

   ```
   Prelude> :t False
   False :: Bool
   ```

   What is the type of the data constructor `MakeExample`? What happens when you request the type of `Example`?

2. What if you try `:info` on `Example` in GHCi? Can you determine what type class instances are defined for the `Example` type using `:info` in GHCi?

3. Try making a new datatype like `Example` but with a single type argument added to `MakeExample`, such as `Int`. What has changed when you query `MakeExample` with `:type` in GHCi?

Unary constructors

In the last section, we asked you to add a single type argument to the `MakeExample` data constructor. In doing so, you changed it from a nullary constructor to a unary one. A unary data constructor takes one argument. In the declaration of the datatype, that parameter will be a type, not a value. Now, instead of your data constructor being a constant, or a known value, the value will be constructed at runtime from the argument we apply it to.

Datatypes that only contain a unary constructor always have the same cardinality as the type they contain. In the following, `Goats` has the same number of inhabitants as `Int`:

```
data Goats = Goats Int deriving (Eq, Show)
```

Anything that is a valid `Int` must also be a valid argument to the `Goats` constructor. Anything that isn't a valid `Int` also isn't a valid count of `Goats`.

For cardinality, this means unary constructors are the identity function.

11.9 newtype

We will now look at a way to define a type that can only ever have a single unary data constructor. We use the `newtype` keyword to mark these types, as they are different from type declarations marked with the `data` keyword as well as from type synonym definitions marked by the `type` keyword. Like other datatypes that have a single unary constructor, the cardinality of a `newtype` is the same as that of the type it contains.

A `newtype` cannot be a product type, sum type, or contain nullary constructors, but it has a few advantages over a vanilla `data` declaration. One is that it has no runtime overhead, as it reuses the representation of the type it contains. It can do this, because it's not allowed to be a record (product type) or tagged union (sum type). The difference between `newtype` and the type it contains is gone by the time the compiler generates the code.

To illustrate, let's say we have a function from `Int -> Bool` for checking whether we have too many goats:

```
tooManyGoats :: Int -> Bool
tooManyGoats n = n > 42
```

We might run into a problem here if we have different limits for different sorts of livestock. What if we mix up the `Int` value of cows where we meant goats? Fortunately, there's a way to address this with unary constructors:

```
newtype Goats =
  Goats Int deriving (Eq, Show)
newtype Cows =
  Cows Int deriving (Eq, Show)
```

Now, we can rewrite our type to be safer, pattern matching in order to access the `Int` inside our data constructor `Goats`:

```
tooManyGoats :: Goats -> Bool
tooManyGoats (Goats n) = n > 42
```

And we can't mix up our livestock counts, either:

```
Prelude> tooManyGoats (Goats 43)
True
Prelude> tooManyGoats (Cows 43)

• Couldn't match expected type ‘Goats’ with
    actual type ‘Cows’
• In the first argument of ‘tooManyGoats’,
    namely ‘(Cows 43)’
  In the expression: tooManyGoats (Cows 43)
  In an equation for ‘it’:
    it = tooManyGoats (Cows 43)
```

Using `newtype` can deliver other advantages related to type class instances. To see these, we need to compare newtypes to type synonyms and regular data declarations. We'll start with a short comparison to type synonyms.

A `newtype` is similar to a type synonym in that the representations of the named type and the type it contains are identical and any distinction between them is stripped away at compile time. So, a

`String` really is a `[Char]`, and `Goats` above is really an `Int`. On the surface, for the human writers and readers of the code, the distinction can be helpful in tracking where data comes from and what it's being used for, but the difference is irrelevant to the compiler.

However, one key contrast between a `newtype` and a type alias is that you can define type class instances for a `newtype` that differs from the instances for its underlying type. You can't do that for type synonyms. Let's take a look at how this works. We'll first define a type class called `TooMany` and an instance for `Int`:

```
class TooMany a where
  tooMany :: a -> Bool

instance TooMany Int where
  tooMany n = n > 42
```

We can use that instance in the REPL but only if we assign the type `Int` to whatever numeric literal we're passing as an argument, because numeric literals are polymorphic. That looks like this:

```
Prelude> tooMany (42 :: Int)
```

Take a moment and play around with this—try leaving off the type declaration and giving it different arguments.

Now, let's say for your goat counting that you want a special instance of `TooMany` that will have different behavior from the `Int` instance. Under the hood, `Goats` is still an `Int`, but the `newtype` declaration will allow you to define a custom instance:

```
newtype Goats = Goats Int deriving Show

instance TooMany Goats where
  tooMany (Goats n) = n > 43
```

Try loading this and passing different arguments to it. Does it behave differently than the `Int` instance above? Do you still need to explicitly assign a type to your numeric literals? What is the type of `tooMany`?

Here, we are able to make the `Goats` newtype have an instance of `TooMany` that has different behavior from the type `Int` that it contains. We can't do this if it's a type synonym. Don't believe us? Try it.

On the other hand, what about the case where we want to reuse the type class instances of the type our newtype contains? For common type classes built into GHC like `Eq`, `Ord`, `Enum`, and `Show`, we get this facility for free, as you've seen with the `deriving` clauses in most datatypes.

For user-defined type classes, we can use a language extension called `GeneralizedNewtypeDeriving`. Language extensions, enabled in GHC by the `LANGUAGE` pragma,[3] tell the compiler to process input in ways beyond what the standard provides for. In this case, this extension will tell the compiler to allow our `newtype` to rely on a type class instance for the type it contains. We can do this, because the representations of the `newtype` and the type it contains are the same. Still, it is outside of the compiler's standard behavior, so we must give it a special instruction to allow us to do this.

First, let's take the case of what we must do without generalized newtype deriving:

```
class TooMany a where
  tooMany :: a -> Bool

instance TooMany Int where
  tooMany n = n > 42

newtype Goats =
  Goats Int deriving (Eq, Show)

instance TooMany Goats where
  tooMany (Goats n) = tooMany n
```

The `Goats` instance will do the same thing as the `Int` instance, but we still have to define it separately.

You can test this yourself to see that they return the same answers.

Now, we'll add the pragma at the top of our source file:

[3] A *pragma* is a special instruction to the compiler placed in source code. The `LANGUAGE` pragma is perhaps more common in GHC Haskell than the other pragmas, but there are other pragmas we will see later in the book.

```
{-# LANGUAGE GeneralizedNewtypeDeriving #-}

class TooMany a where
  tooMany :: a -> Bool

instance TooMany Int where
  tooMany n = n > 42

newtype Goats =
  Goats Int deriving (Eq, Show, TooMany)
```

We don't have to define an instance of `TooMany` for `Goats` that's identical to the `Int` instance. We can reuse the instance that we already have.

This is also nice for times when we want every type class instance to be the same *except* for the one we want to change.

Exercises: Logic goats

1. Reusing the `TooMany` type class, write an instance of the type class for the type `(Int, String)`. This will require adding a language pragma named `FlexibleInstances`[4] *if* you do not use a newtype—GHC will tell you what to do.
2. Make another `TooMany` instance for `(Int, Int)`. Sum the values together under the assumption that this is a count of goats from two fields.
3. Make another `TooMany` instance, this time for `(Num a, TooMany a) => (a, a)`. This can mean whatever you want, such as summing the two numbers together.

11.10 Sum types

Now that we've looked at data constructor arities, we're ready to define the algebra of algebraic datatypes. The first that we'll look at is the *sum type*, and our first example of one is `Bool`:

```
data Bool = False | True
```

[4] `https://ghc.haskell.org/trac/haskell-prime/wiki/FlexibleInstances`

We mentioned previously that the | represents logical disjunction, that is, "or." This is the *sum* in algebraic datatypes. To know the cardinality of sum types, we *add* the cardinalities of their data constructors. `True` and `False` take no type arguments and thus are nullary constructors, each with a value of 1.

Now, we do some arithmetic. As we said earlier, nullary constructors are 1, and sum types are +, or addition, when we are talking about cardinality:

```
data Bool = False | True
```

How many values inhabit `Bool`? There are two data constructors, each representing only one possible value. Given that the | syntax represents addition:

```
-- ?? represents the cardinality
True | False = ??


True + False == ??


-- False and True both == 1
1 + 1 == ??
```

We see that the cardinality of `Bool` is:

```
1 + 1 == 2


-- List of all possible values for Bool
[True, False] -- length is 2
```

You can check this in your REPL:

```
Prelude> length (enumFrom False)
2
```

From the above, we see that when working with a `Bool`, we must reason about two possible values. Sum types are a way of expressing alternative possibilities within a single datatype.

Signed 8-bit hardware integers in Haskell are defined using the aforementioned `Int8` datatype with a range of values from -128 to 127. It's not defined this way, but you could think of it as a sum type of the numbers in that range, leading to the cardinality of 256 that we saw.

Exercises: Pity the Bool

1. Given a datatype:

```
data BigSmall =
    Big Bool
  | Small Bool
  deriving (Eq, Show)
```

What is its cardinality? Hint: We already know `Bool`'s cardinality. Show your work, as demonstrated earlier.

2. Given a datatype:

```
-- bring Int8 in scope
import Data.Int

data NumberOrBool =
    Numba Int8
  | BoolyBool Bool
  deriving (Eq, Show)

myNumba = Numba (-128)
```

What is the cardinality of `NumberOrBool`? What happens if you try to create a `Numba` with a numeric literal larger than 127? And with a numeric literal smaller than -128?

If you choose `(-128)` for a value precisely, you'll notice you get a spurious warning:

```
Prelude> n = Numba (-129)
Literal -129 is out of the
  Int8 range -128..127
```

Now, since -128 is a perfectly valid `Int8` value, you could choose to ignore this. What happens is that `(-128)` desugars into `(negate 128)`. The compiler sees that you expect the type `Int8`, but `Int8`'s max boundary is 127. So even though you're negating 128, it hasn't done that step yet and *immediately* whines about 128 being larger than 127. One way to avoid that warning is the following:

```
Prelude> n = (-128)
Prelude> x = Numba n
```

Or you can use the `NegativeLiterals` extension as it recommends:

```
Prelude> :set -XNegativeLiterals
Prelude> n = Numba (-128)
```

Note that the negative literals extension doesn't prevent the warning if you use `negate` directly.

11.11 Product types

What does it mean for a type to be a product? A product type's cardinality is the *product* of the cardinalities of its inhabitants. Arithmetically, products are the result of *multiplication*. Whereas a sum type expresses *or*, a product type expresses *and*.

For those who have programmed in C-like languages before, a product is like a `struct`. For those who haven't, a product is a way to carry multiple values around in a single data constructor. Any data constructor with two or more type arguments is a product.

We said previously that tuples are anonymous products. The declaration of the tuple type looks like this:

```
(,) :: a -> b -> (a, b)
```

This datatype is a product, like a product type: it gives you a way to encapsulate two pieces of data in a single value. Those two pieces of data can be different types, but they don't have to be.

We'll look next at a somewhat silly sum type:

```
data QuantumBool = QuantumTrue
                 | QuantumFalse
                 | QuantumBoth
                 deriving (Eq, Show)
```

What is the cardinality of this sum type?

For reasons that might be obvious, a cardinality of 2 makes it harder to show the difference between sum and product cardinality, so `QuantumBool` has a cardinality of 3. Now we're going to define a product type that contains two `QuantumBool` values:

```
data TwoQs =
  MkTwoQs QuantumBool QuantumBool
  deriving (Eq, Show)
```

The datatype `TwoQs` has one data constructor, `MkTwoQs`, that takes two arguments, making it a product of the two types that inhabit it. Each argument is of type `QuantumBool`, which has a cardinality of 3.

You can write this out to help you visualize it, if you like. A `MkTwoQs` value could be:

```
-- Just 4 examples, there are more
MkTwoQs QuantumTrue QuantumTrue
MkTwoQs QuantumTrue QuantumFalse
MkTwoQs QuantumTrue QuantumBoth
MkTwoQs QuantumFalse QuantumFalse
```

Note that there is no special syntax denoting product types as there is with sums and `|`. `MkTwoQs` is a data constructor taking two type arguments, which both happen to be the same type. It is a product type, the product of two `QuantumBools`. The number of potential values that can manifest in this type is the cardinality of one of its type arguments times the cardinality of the other. So, what is the cardinality of `TwoQs`?

We could also have written the `TwoQs` type using a *type alias* and the tuple data constructor. Type aliases create type constructors, not data constructors:

```
type TwoQs = (QuantumBool, QuantumBool)
```

The cardinality of this will be the same as it was previously.

The reason it's important to understand cardinality is that the cardinality of a datatype roughly equates to how difficult it is to reason about.

Record syntax

Records in Haskell are product types with additional syntax to provide convenient accessors to fields within a record. Let's begin by defining a simple product type:

```
data Person =
  MkPerson String Int
  deriving (Eq, Show)
```

That is the familiar product type structure: the `MkPerson` data constructor takes two type arguments in its definition, a `String` value (a name) and an `Int` value (an age). The cardinality of this is frankly terrifying.

As we've seen in previous examples, we can unpack the contents of this type using functions that return what we want from our little box of values:

```
-- sample data
jm = MkPerson "julie" 108
ca = MkPerson "chris" 16

namae :: Person -> String
namae (MkPerson s _) = s
```

If you use the `namae` function in your REPL, it will return the `String` value from your data.

Let's see how we could define a similar product type but with record syntax:

```
data Person =
  Person { name :: String
         , age :: Int }
         deriving (Eq, Show)
```

You can see the similarity to the `Person` type defined above, but defining it as a record means there are now named record field accessors. They're just functions that go from the product type to a member of product:

```
Prelude> :t name
name :: Person -> String
Prelude> :t age
age :: Person -> Int
```

You can use this directly in GHCi:

```
Prelude> Person "Papu" 5
Person {name = "Papu", age = 5}

Prelude> papu = Person "Papu" 5
Prelude> age papu
5
Prelude> name papu
"Papu"
```

You can also do it from data that is in a file. Change the `jm` and `ca` data above so that it is now of type `Person`, reload your source file, and try using the record field accessors in GHCi to query the values.

11.12 Normal form

We've looked at the algebra behind Haskell's algebraic datatypes, and explored how this is useful for understanding the cardinality of datatypes. But the algebra doesn't stop there. All the existing algebraic rules for products and sums apply in type systems, and that includes the *distributive* property. Let's take a look at how that works in arithmetic:

```
2 * (3 + 4)
  2 * (7)
    14
```

We can rewrite this with the multiplication distributed over the addition and obtain the same result:

```
2 * 3 + 2 * 4
  (6) + (8)
     14
```

This is known as a "sum of products." In normal arithmetic, the expression is in normal form when it's been reduced to a final result. However, if you think of the numerals in the above expressions as representations of set cardinality, then the sum of products expression is in normal form, as there is no computation to perform.

The distributive property can be generalized as follows:

```
a * (b + c) -> (a * b) + (a * c)
```

And this is true of Haskell's types as well! Product types distribute over sum types. To play with this, we'll first define some datatypes:

```
data Fiction = Fiction deriving Show
data Nonfiction = Nonfiction deriving Show

data BookType = FictionBook Fiction
              | NonfictionBook Nonfiction
              deriving Show
```

We define two types with only single, nullary inhabitants: `Fiction` and `Nonfiction`. The reasons for doing that may not be immediately clear, but recall that we said you can't use a type while only permitting one of its inhabitants as a possible value. You can't ask for a value of type `Bool` while declaring in your types that it must always be `True`—you must admit the possibility of either `Bool` value. So, declaring the `Fiction` and `Nonfiction` types will allow us to factor out the book types (below).

Then, we have a sum type, `BookType`, with constructors that take the `Fiction` and `Nonfiction` *types* as arguments. It's important to remember that, although the type constructors and data constructors of `Fiction` and `Nonfiction` have the same name, they are not the same, and it is the type constructors that are the arguments to `FictionBook` and `NonfictionBook`. Take a moment and rename them to demonstrate this to yourself.

So, we have our sum type. Next, we're going to define a type synonym called `AuthorName` and a product type called `Author`. The type synonym doesn't really do anything except help us keep track of which `String` we're using in the `Author` type:

```
type AuthorName = String

data Author = Author (AuthorName, BookType)
```

This isn't a sum of products, so it isn't normal form. It can, in some sense, be evaluated to tease apart the values that are hiding in the sum type, `BookType`. Again, we can apply the distributive property and rewrite `Author` in normal form:

```
type AuthorName = String

data Author =
    Fiction AuthorName
  | Nonfiction AuthorName
  deriving (Eq, Show)
```

Products distribute over sums. Just as we would do with the expression `a * (b + c)`, where the inhabitants of the sum type `BookType` are the `b` and `c`, we break those values out and make a sum of products. Now it's in normal form, because no further evaluation can be done of these constructors until some operation or computation is done using these types.

Another example of normal form can be found in the `Expr` type, which is common in papers about type systems and programming languages:

```
data Expr =
    Number Int
  | Add Expr Expr
  | Minus Expr
  | Mult Expr Expr
  | Divide Expr Expr
```

This is in normal form, because it's a sum (type) of products: `(Number Int) + Add (Expr Expr) + ...` and so on.

A stricter interpretation of normal form or the "sum of products" would require representing products with tuples and sums with `Either`. The previous datatype in that form would look like the following:

```
type Number = Int
type Add = (Expr, Expr)
type Minus = Expr
type Mult = (Expr, Expr)
type Divide = (Expr, Expr)
```

```
type Expr =
  Either Number
    (Either Add
      (Either Minus
        (Either Mult Divide)))
```

This representation finds applications in problems where one is writing functions or *folds* over the representations of datatypes, such as with generics and metaprogramming. Some of these methods have their application in Haskell but should be used judiciously and aren't always easy to work with.

The `Either` type will be explained in detail in the next chapter.

Exercises: How does your garden grow?

1. Given the type:

```
data FlowerType = Gardenia
                | Daisy
                | Rose
                | Lilac
                deriving Show

type Gardener = String

data Garden =
  Garden Gardener FlowerType
  deriving Show
```

What is the sum of products normal form of `Garden`?

11.13 Constructing and deconstructing values

There are essentially two things we can do with a value: we can generate or construct it, or we can match on it and consume it. We discussed above about why data and type constructors are called *constructors*, and this section will elaborate on that and how to construct values of different types. You have already been doing this in previous chapters, but we hope this section will lead you to a deeper understanding.

Construction and deconstruction of values form a duality. Data is immutable in Haskell, so values carry with them the information about how they were created. We can use that information when we consume or deconstruct the value.

We'll start by defining a collection of datatypes:

```
data GuessWhat =
  ChickenButt deriving (Eq, Show)

data Id a =
  MkId a deriving (Eq, Show)

data Product a b =
  Product a b deriving (Eq, Show)

data Sum a b =
    First  a
  | Second b
  deriving (Eq, Show)

data RecordProduct a b =
  RecordProduct { pfirst  :: a
                , psecond :: b }
                deriving (Eq, Show)
```

Now that we have different sorts of datatypes to work with, we'll move on to constructing values of those types.

Sum and Product

`Sum` and `Product` are ways to represent arbitrary sums and products in types. In ordinary Haskell code, it's unlikely you'd need or want nestable sums and products unless you're doing something fairly advanced, but we're using them in this case as a means of demonstration.

If you have two values in a product, then the conversion to using `Product` is straightforward (n.b.: The `Sum` and `Product` declarations from above will need to be in scope for all of the following examples):

```
newtype NumCow =
  NumCow Int
  deriving (Eq, Show)

newtype NumPig =
  NumPig Int
  deriving (Eq, Show)

data Farmhouse =
  Farmhouse NumCow NumPig
  deriving (Eq, Show)

type Farmhouse' = Product NumCow NumPig
```

`Farmhouse` and `Farmhouse'` are the same.

For an example with three values in the product instead of two, we must begin to take advantage of the fact that `Product` takes two arguments, one of which can also be another `Product` of values. In fact, you can nest them as far as you can stomach or until the compiler chokes:

```
newtype NumSheep =
  NumSheep Int
  deriving (Eq, Show)

data BigFarmhouse =
  BigFarmhouse NumCow NumPig NumSheep
  deriving (Eq, Show)

type BigFarmhouse' =
  Product NumCow (Product NumPig NumSheep)
```

We can perform a similar trick with `Sum`:

```
type Name = String
type Age = Int
type LovesMud = Bool
```

Sheep can produce between 2 and 30 pounds (0.9 and 13 kilos) of wool per year! Icelandic sheep don't produce as much wool per year as other breeds, but the wool they do produce is finer:

```
type PoundsOfWool = Int

data CowInfo =
  CowInfo Name Age
  deriving (Eq, Show)

data PigInfo =
  PigInfo Name Age LovesMud
  deriving (Eq, Show)

data SheepInfo =
  SheepInfo Name Age PoundsOfWool
  deriving (Eq, Show)

data Animal =
    Cow CowInfo
  | Pig PigInfo
  | Sheep SheepInfo
  deriving (Eq, Show)

-- Alternatively
type Animal' =
  Sum CowInfo (Sum PigInfo SheepInfo)
```

Again, in the REPL, we use `First` and `Second` to pattern match on the data constructors of `Sum`. The first time, we'll get it right:

```
Prelude> bess' = (CowInfo "Bess" 4)
Prelude> bess = First bess' :: Animal'

Prelude> :{
*Main| let e' =
*Main|       Second (SheepInfo "Elmer" 5 5)
*Main| :}
Prelude> elmer = Second e' :: Animal'
```

This time, we'll make a mistake:

```
Prelude> :{
*Main| let elmo' =
```

```
*Main|       Second (SheepInfo "Elmo" 5 5)
*Main| :}
Prelude> elmo = First elmo' :: Animal'

• Couldn't match type ‘Sum a0 SheepInfo’
    with ‘CowInfo’
  Expected type: Animal'
    Actual type: Sum (Sum a0 SheepInfo)
      (Sum PigInfo SheepInfo)
• In the expression: First elmo' :: Animal'
  In an equation for ‘elmo’: elmo =
    First elmo' :: Animal'
```

The first data constructor, `First`, has the argument `CowInfo`, but `SheepInfo` is nested within the `Second` constructor (it is the `Second` of the `Second`). We can see how they don't match, and the mistaken attempt nests in the wrong direction:

```
Prelude> sheep = SheepInfo "Baaaaa" 5 5
Prelude> :t First (Second sheep)
First (Second (SheepInfo "Baaaaa" 5 5))
  :: Sum (Sum a SheepInfo) b

Prelude> :info Animal'
type Animal' =
  Sum CowInfo (Sum PigInfo SheepInfo)
  -- Defined at code/animalFarm1.hs:61:1
```

As we said, the actual types `Sum` and `Product` themselves aren't used very often in standard Haskell code, but they can be useful for developing an intuition about the structure of sum and product types.

Constructing values

Our first datatype, `GuessWhat`, is trivial, equivalent to the `()` unit type:

```
trivialValue :: GuessWhat
trivialValue = ChickenButt
```

Types like this are sometimes used to signal discrete concepts that you don't want to flatten into the unit type. We'll elaborate on how this can make code easier to understand or better abstracted later. There is nothing special in the syntax here. We define `trivialValue` to be the nullary data constructor `ChickenButt`, and we have a value of type `GuessWhat`.

Next, we look at a unary type constructor that contains one unary data constructor:

```
data Id a =
  MkId a deriving (Eq, Show)
```

Because `Id` has an argument, we have to apply it to something before we can construct a value of that type:

```
idInt :: Id Integer
idInt = MkId 10
```

We turn our attention to our product type with two arguments. We're going to define some type synonyms first to make this more readable:

```
type Awesome = Bool
type Name = String

person :: Product Name Awesome
person = Product "Simon" True
```

The type synonyms `Awesome` and `Name` here are for clarity. They don't obligate us to change our *terms*. We could have used datatypes instead of type synonyms, as we will in the sum type example below, but this is a quick and painless way to construct the value that we need. Notice that we're relying on the `Product` data constructor that we defined above. The `Product` data constructor is a function of two arguments, `Name` *and* `Awesome`. Notice, also, that Simons are invariably awesome.

Now, we'll use the `Sum` type defined above:

```
data Sum a b =
    First  a
  | Second b
  deriving (Eq, Show)

data Twitter =
  Twitter deriving (Eq, Show)

data AskFm =
  AskFm deriving (Eq, Show)

socialNetwork :: Sum Twitter AskFm
socialNetwork = First Twitter
```

Here, our type is a sum of `Twitter` *or* `AskFm`. We don't have both values at the same time without the use of a product, because sums are a means of expressing disjunction or the ability to have one of several possible values. We have to use one of the data constructors generated by the definition of `Sum` in order to indicate which of the possibilities in the disjunction we mean to express. Consider the case where we mix them up:

```
Prelude> type SN = Sum Twitter AskFm
Prelude> Second Twitter :: SN

• Couldn't match type ‘Twitter’
    with ‘AskFm’
  Expected type: SN
    Actual type: Sum Twitter Twitter
• In the expression: Second Twitter :: SN
  In an equation for ‘it’: it =
    Second Twitter :: SN

Prelude> First AskFm :: Sum Twitter AskFm

• Couldn't match type ‘AskFm’
    with ‘Twitter’
  Expected type: Sum Twitter AskFm
```

```
    Actual type: Sum AskFm AskFm
• In the expression:
    First AskFm :: Sum Twitter AskFm
  In an equation for ‘it’: it =
    First AskFm :: Sum Twitter AskFm
```

The appropriate assignment of types to specific constructors is dependent on the assertions in the type. The type signature `Sum Twitter AskFm` tells you which goes with the data constructor `First` and which goes with the data constructor `Second`. We can assert that ordering directly by writing a datatype like this:

```
data SocialNetwork =
    Twitter
  | AskFm
  deriving (Eq, Show)
```

The data constructors for `Twitter` and `AskFm` are direct inhabitants of the sum type `SocialNetwork`, where before they inhabited the `Sum` type. Let's consider how this might look with type synonyms:

```
type Twitter = String
type AskFm = String

twitter :: Sum Twitter AskFm
twitter = First "Twitter"

askfm :: Sum Twitter AskFm
askfm = First "AskFm"
```

There's a problem with the above example. The name of `askfm` implies that we mean `Second "AskFm"`, but we messed up. Because we use type synonyms instead of defining datatypes, the type system doesn't catch the mistake. The type checker has no way of knowing we made a mistake, because both values are type `String`. Try to avoid using type synonyms with unstructured data like text or binary. Type synonyms are best used when you want something lighter weight than newtypes but also want your type signatures to be more explicit.

Finally, we'll consider the product that uses record syntax:

```
Prelude> :t RecordProduct
RecordProduct :: a
              -> b
              -> RecordProduct a b

Prelude> :t Product
Product :: a -> b -> Product a b
```

The first thing to notice is that you can construct values of products that use record syntax in a manner identical to that of non-record products. Records are just syntax to create field references. They don't do much heavy lifting in Haskell, but they are convenient:

```
myRecord :: RecordProduct Integer Float
myRecord = RecordProduct 42 0.00001
```

We can take advantage of the fields that we define on our record to construct values in a slightly different style. This can be convenient for making things a little more obvious:

```
myRecord :: RecordProduct Integer Float
myRecord =
  RecordProduct { pfirst = 42
                , psecond = 0.00001 }
```

This is a bit more compelling when you have domain-specific names for things:

```
data OperatingSystem =
       GnuPlusLinux
     | OpenBSDPlusNevermindJustBSDStill
     | Mac
     | Windows
     deriving (Eq, Show)

data ProgLang =
       Haskell
     | Agda
     | Idris
     | PureScript
     deriving (Eq, Show)
```

```
data Programmer =
  Programmer { os :: OperatingSystem
             , lang :: ProgLang }
  deriving (Eq, Show)
```

Then, we can construct a value from the record product `Programmer`:

```
Prelude> :t Programmer
Programmer :: OperatingSystem
           -> ProgLang
           -> Programmer

nineToFive :: Programmer
nineToFive = Programmer { os = Mac
                        , lang = Haskell }
```

We can re-order field assignments when we use record syntax:

```
feelingWizardly :: Programmer
feelingWizardly =
  Programmer { lang = Agda
             , os = GnuPlusLinux }
```

Exercise: Programmers

Write a function that generates all possible values of `Programmer`. Use the provided lists of inhabitants of `OperatingSystem` and `ProgLang`:

```
allOperatingSystems :: [OperatingSystem]
allOperatingSystems =
  [ GnuPlusLinux
  , OpenBSDPlusNevermindJustBSDStill
  , Mac
  , Windows
  ]

allLanguages :: [ProgLang]
allLanguages =
  [Haskell, Agda, Idris, PureScript]
```

```
allProgrammers :: [Programmer]
allProgrammers = undefined
```

`Programmer` is a product of two types. You can determine the number of inhabitants of `Programmer` by calculating:

```
  length allOperatingSystems
* length allLanguages
```

This is the essence of how product types and the number of inhabitants relate.

There are several ways you could write a function to do this, and some may produce a list that has duplicate values in it. If your resulting list has duplicate values in it, you can use `nub` from `Data.List` to remove duplicate values over your `allProgrammers` value. Either way, if your result (minus any duplicate values) equals the number returned by multiplying those lengths together, you've probably got it figured out. Try to be clever and make it work without manually typing out the values.

Accidental bottoms from records

We're going to reuse the previous `Programmer` datatype to see what happens if we construct a value using record syntax but forget a field:

```
Prelude> :{
*Main| let partialAf =
*Main|       Programmer {os = GnuPlusLinux}
*Main| :}

• Fields of ‘Programmer’ not initialised:
    lang
• In the expression:
    Programmer {os = GnuPlusLinux}
  In an equation for ‘partialAf’:
      partialAf =
        Programmer {os = GnuPlusLinux}
```

And if we don't heed this warning:

```
Prelude> partialAf
Programmer {os = GnuPlusLinux, lang =
*** Exception:
      Missing field in
      record construction lang
```

Do *not* do this in your code! Either define the whole record at once or not at all. If you think you need this, your code needs to be refactored. Partial application of the data constructor suffices to handle this:

```
-- Works the same as if
-- we'd used record syntax.

data ThereYet =
  There Float Int Bool
  deriving (Eq, Show)

nope = There

-- Who needs a "builder pattern"?
notYet :: Int -> Bool -> ThereYet
notYet = nope 25.5

notQuite :: Bool -> ThereYet
notQuite = notYet 10

yusssss :: ThereYet
yusssss = notQuite False

-- Not I, said the Haskell user.
```

Notice the way our types progress:

```
There    :: Float -> Int -> Bool -> ThereYet
notYet   ::          Int -> Bool -> ThereYet
notQuite ::                 Bool -> ThereYet
yusssss  ::                         ThereYet
```

Percolate *values* through your programs, not bottoms.

Deconstructing values

When we discussed folds, we mentioned the idea of the catamorphism. We explained that a catamorphism is about *deconstructing* lists. This idea is generally applicable to any datatype that has values. Now that we've thoroughly explored constructing values, the time has come to destroy what we have built. Wait, no—we mean deconstruct.

We begin, as always, with some datatypes:

```
newtype Name    = Name String deriving Show
newtype Acres   = Acres Int deriving Show

-- FarmerType is a Sum
data FarmerType = DairyFarmer
                | WheatFarmer
                | SoybeanFarmer
                deriving Show

-- Farmer is a plain old product of
-- Name, Acres, and FarmerType
data Farmer =
  Farmer Name Acres FarmerType
  deriving Show
```

Now, we're going to write a very basic function that breaks down and unpacks the data inside our constructors:

```
isDairyFarmer :: Farmer -> Bool
isDairyFarmer (Farmer _ _ DairyFarmer) =
  True
isDairyFarmer _ =
  False
```

`DairyFarmer` is one value of the `FarmerType` type that is packed up inside our `Farmer` product type. But our function can pull that value out, pattern match on it, and tell us just what we're looking for.

Here's an alternative formulation with a product that uses record syntax:

```
data FarmerRec =
  FarmerRec { name       :: Name
            , acres      :: Acres
            , farmerType :: FarmerType }
            deriving Show

isDairyFarmerRec :: FarmerRec -> Bool
isDairyFarmerRec farmer =
  case farmerType farmer of
    DairyFarmer -> True
    _           -> False
```

This is just another way of unpacking or deconstructing the contents of a product type.

Accidental bottoms from records

We take bottoms very seriously. You can easily propagate bottoms through record types, and we implore you not to do so. Please, do not do this:

```
data Automobile = Null
                | Car { make :: String
                      , model :: String
                      , year :: Integer }
                deriving (Eq, Show)
```

This is a terrible thing to do, for a couple of reasons. One is this `Null` nonsense. Haskell offers you the perfectly lovely datatype `Maybe`, which you should use, instead. Secondly, consider the case where you have a `Null` value, but you've used one of the record accessors:

```
Prelude> make Null
"*** Exception: No match in
                record selector make
```

Don't do this.

How do we fix it? Well, first, whenever we have a product that uses record accessors, keep it separate from any sum type that is wrapping it. To do this, split out the product into an independent type with its

own type constructor instead of only as an inline data constructor product:

```
-- Split out the record/product
data Car = Car { make :: String
               , model :: String
               , year :: Integer }
               deriving (Eq, Show)

-- The Null is still not great, but
-- we're leaving it in to make a point
data Automobile = Null
                | Automobile Car
                deriving (Eq, Show)
```

Now, if we attempt to do something silly, the type system catches us:

```
Prelude> make Null

• Couldn't match expected type ‘Car’ with
    actual type ‘Automobile’
• In the first argument of ‘make’,
    namely ‘Null’
  In the expression: make Null
  In an equation for ‘it’: it = make Null
```

In Haskell, we want the type checker to catch us doing things wrong, so we can fix our mistakes *before* problems multiply and things go wrong at runtime. But the type checker can best help those who help themselves.

11.14 Function type is exponential

In the arithmetic of calculating inhabitants of types, the function type is the exponent operator. Given a function `a -> b`, we can calculate the inhabitants with the formula b^a.

So if `b` and `a` are `Bool`, then 2^2 is how you could express the number of inhabitants in a function of `Bool -> Bool`. Similarly, a function of

`Bool` to something of 3 inhabitants would be 3^2 and thus have nine possible implementations:

```
a -> b -> c
(c ^ b) ^ a
```

Given the laws of arithmetic, this can be rewritten as:

```
c ^ (b * a)
```

Earlier, we identified the type `(Bool, Bool)` as having four inhabitants. This can be determined by either writing out all the possible unique inhabitants or, more easily, by doing the arithmetic of `(1 + 1) * (1 + 1)`. Next, we'll see that the type of functions `(->)` is, in the algebra of types, the exponentiation operator. We'll use a datatype with three cases, because `Bool` has one difficulty: two plus two, two times two, and two to the power of two all equal the same thing. Let's review the arithmetic of sum types:

```
data Quantum =
    Yes
  | No
  | Both
  deriving (Eq, Show)

-- 3 + 3
quantSum1 :: Either Quantum Quantum
quantSum1 = Right Yes

quantSum2 :: Either Quantum Quantum
quantSum2 = Right No

quantSum3 :: Either Quantum Quantum
quantSum3 = Right Both

quantSum4 :: Either Quantum Quantum
quantSum4 = Left Yes
-- You can fill in the next two.
```

And now the arithmetic of product types:

```
-- 3 * 3
quantProd1 :: (Quantum, Quantum)
quantProd1 = (Yes, Yes)

quantProd2 :: (Quantum, Quantum)
quantProd2 = (Yes, No)

quantProd3 :: (Quantum, Quantum)
quantProd3 = (Yes, Both)

quantProd4 :: (Quantum, Quantum)
quantProd4 = (No, Yes)

quantProd5 :: (Quantum, Quantum)
quantProd5 = (No, No)

quantProd6 :: (Quantum, Quantum)
quantProd6 = (No, Both)

quantProd7 :: (Quantum, Quantum)
quantProd7 = (Both, Yes)
-- You can determine the final two.
```

And now a function type. Each possible unique implementation of the function is an inhabitant:

```
-- 3 ^ 3

quantFlip1 :: Quantum -> Quantum
quantFlip1 Yes  = Yes
quantFlip1 No   = Yes
quantFlip1 Both = Yes

quantFlip2 :: Quantum -> Quantum
quantFlip2 Yes  = Yes
quantFlip2 No   = Yes
quantFlip2 Both = No
```

```
quantFlip3 :: Quantum -> Quantum
quantFlip3 Yes  = Yes
quantFlip3 No   = Yes
quantFlip3 Both = Both

quantFlip4 :: Quantum -> Quantum
quantFlip4 Yes  = Yes
quantFlip4 No   = No
quantFlip4 Both = Yes

quantFlip5 :: Quantum -> Quantum
quantFlip5 Yes  = Yes
quantFlip5 No   = Both
quantFlip5 Both = Yes
```

You can figure out the remaining possibilities yourself.

Exponentiation in what order?

Consider the following function:

```
convert :: Quantum -> Bool
convert = undefined
```

According to the equality of `a -> b` and b^a, there should be 2^3 or 8 implementations of this function. Does this hold? Write it out, and prove it for yourself.

Exercises: The Quad

Determine how many unique inhabitants each type has.

Suggestion: do the arithmetic, unless you want to verify. Writing them out gets tedious quickly:

1.
```
data Quad =
    One
  | Two
  | Three
  | Four
  deriving (Eq, Show)

-- How many different forms can this take?
eQuad :: Either Quad Quad
eQuad = ???
```

2.
```
prodQuad :: (Quad, Quad)
```

3.
```
funcQuad :: Quad -> Quad
```

4.
```
prodTBool :: (Bool, Bool, Bool)
```

5.
```
gTwo :: Bool -> Bool -> Bool
```

6. Hint: five digit number

```
fTwo :: Bool -> Quad -> Quad
```

11.15 Higher-kinded datatypes

You may recall that we discussed kinds earlier in this chapter. Kinds are the types of type constructors, primarily encoding the number of arguments they take. The default kind in Haskell is `*`. Kind signatures work like type signatures, using the same `::` and `->` syntax, but there are only a few kinds, and you'll most often see `*`.

Kinds are not types until they are fully applied. Only types have inhabitants at the term level. The kind `* -> *` is waiting for a single `*` before it is fully applied. The kind `* -> * -> *` must be applied twice before it will be a real type. This is known as a *higher-kinded type*. Lists, for example, are higher-kinded datatypes in Haskell.

Because types can be generically polymorphic by taking type arguments, they can be applied at the type level:

```
-- identical to (a, b, c, d)
data Silly a b c d =
  MkSilly a b c d deriving Show
```

In GHCi:

```
Prelude> :kind Silly
Silly :: * -> * -> * -> * -> *

Prelude> :kind Silly Int
Silly Int :: * -> * -> * -> *

Prelude> :kind Silly Int String
Silly Int String :: * -> * -> *

Prelude> :kind Silly Int String Bool
Silly Int String Bool :: * -> *

Prelude> :kind Silly Int String Bool String
Silly Int String Bool String :: *

-- Identical to (a, b, c, d)
Prelude> :kind (,,,)
(,,,) :: * -> * -> * -> * -> *

Prelude> :kind (Int, String, Bool, String)
(Int, String, Bool, String) :: *
```

Getting comfortable with higher-kinded types is important, as type arguments provide a generic way to express a "hole" to be filled by consumers of your datatype later. Take the following as an example from a library one of the authors maintains, called Bloodhound:[5]

```
data EsResultFound a =
  EsResultFound
  { _version :: DocVersion
  , _source  :: a
  } deriving (Eq, Show)
```

[5]You can find it here: `http://hackage.haskell.org/package/bloodhound`. If you are not a programmer and do not know what Elasticsearch and JSON are, try not to worry too much about the specifics. Elasticsearch is a search engine, and JSON is a format for transmitting data, especially between servers and web applications.

We know that this particular kind of response from Elasticsearch will include a `DocVersion` value, so that's been assigned a type. On the other hand, `_source` has type `a`, because we have no idea what the structure of the documents they're pulling from Elasticsearch look like. In practice, we do need to be able to do *something* with that value of type `a`. The thing we will want to do with it—the way we will consume or use that data—will usually be a `FromJSON` type class instance for deserializing JSON data into a Haskell datatype. But in Haskell, we do not conventionally put constraints on datatypes. That is, we don't want to constrain that polymorphic `a` in the datatype. The `FromJSON` type class will likely (assuming that's what is needed in a given context) constrain the variable in the type signature(s) for the function(s) that will process this data.

Accordingly, the `FromJSON` type class instance for `EsResultFound` requires a `FromJSON` instance for that `a`:

```
instance (FromJSON a) =>
  FromJSON (EsResultFound a) where
    parseJSON (Object v) =
          EsResultFound
      <$> v .: "_version"
      <*> v .: "_source"
    parseJSON _ = empty
```

As you can hopefully see from this, by not fully applying the type—by leaving it higher-kinded—space is left for the type of the response to vary, for the "hole" to be filled in by the end user.

11.16 Lists are polymorphic

What makes a list polymorphic? In what way can it take many forms? What makes them polymorphic is that lists in Haskell can contain values of any type. You do not have an `a` until the list type's type argument has been fully applied:

```
data  []    a   =   []  |  a    :   [a]
--    [1]  [2]      [3]    [4]  [5]  [6]
```

1. The type constructor for lists has special `[]` syntax.

2. Single type argument to `[]`. This is the type of value our list contains.
3. Nil/empty list value constructor, again with the special `[]` syntax. `[]` marks the end of the list.
4. A single value of type `a`.
5. `:` is an infix data constructor. It is a product of `a` and `[a]`.
6. The rest of the list.

Infix type and data constructors When we give an operator a non-alphanumeric name, it is infix by default. For example, all the non-alphanumeric arithmetic functions are infix operators, while we have some alphanumeric arithmetic functions, such as `div` and `mod` that are prefix by default. So far, we've only seen alphanumeric data constructors, except for the cons constructor in the list type, but the same rule applies to them.

Any operator that starts with a colon (`:`) must be an infix type or data constructor. All infix data constructors must start with a colon. The type constructor of functions, `->`, is the only infix type constructor that doesn't start with a colon. Another exception is that they cannot be `::`, as this syntax is reserved for type assertions.

In the following example, we'll define the list type without using an infix constructor:

```
-- Same type, redefined
-- with different syntax
data List  a  =  Nil | Cons  a   (List a)
--   [1]  [2]    [3]   [5]  [4]  [6]
```

1. The `List` type constructor.
2. The `a` type parameter to `List`.
3. `Nil` is the empty list value, which also marks the end of a list.
4. A single value of type `a` in the `Cons` product.
5. The `Cons` constructor, product of `a` and `List a`.

6. The rest of the list.

How do we use our `List` type?

```
Prelude> nil = Nil
Prelude> :t nil
nil :: List a
```

The type parameter isn't applied, because `Nil` by itself doesn't tell the type checker what the `List` contains. But if we give it some information, then the `a` can be assigned a concrete type:

```
Prelude> oneItem = (Cons "woohoo!" Nil)
Prelude> :t oneItem
oneItem :: List [Char]
```

And how are our list types kinded?

```
Prelude> :kind List
List :: * -> *
Prelude> :kind []
[] :: * -> *

Prelude> :kind List Int
List Int :: *
Prelude> :kind [Int]
[Int] :: *
```

Much as we can refer to the function `not` before we've applied its argument, we can refer to the list type constructor, `[]`, before we've applied it to a type argument:

```
Prelude> :t not
not :: Bool -> Bool
Prelude> :t not True
not True :: Bool

Prelude> :k []
[] :: * -> *
Prelude> :k [Int]
[Int] :: *
```

The difference is that the argument of `not` is any value of type `Bool`, and the argument of `[]` is any type of kind `*`. So, they're similar, but type constructors are functions one level up, structuring things that cannot exist at runtime—it's purely static and describes the structure of your types.

11.17 Binary tree

Now we turn our attention to a type similar to list. The type constructor for binary trees can take an argument, and it is also recursive, like lists:

```
data BinaryTree a =
    Leaf
  | Node (BinaryTree a) a (BinaryTree a)
  deriving (Eq, Ord, Show)
```

This tree has a value of type `a` at each node. Each node could be a terminal node, called a *leaf*, or it could branch and have two subtrees. The subtrees are also of type `BinaryTree a`, so this type is recursive. Each binary tree can store yet another binary tree, which allows for trees of arbitrary depth.

In some cases, binary trees can be more efficient for structuring and accessing data than a list, especially if you know how to order your values in a way that lets you know whether to look "left" or "right" to find what you want. On the other hand, a tree that only branches to the right is indistinguishable from an ordinary list. For now, we won't concern ourselves too much with this, as we'll talk about the proper application of data structures later. Instead, you're going to write some functions for processing `BinaryTree` values.

Inserting into trees

The first thing to be aware of is that we need `Ord` in order to have enough information about our values to know how to arrange them in our tree. Accordingly, if something is lower, we want to insert it somewhere on the left-hand part of our tree. If it's greater than the current node value, it should go somewhere to the right. Left lesser, right greater is a common convention for arranging binary

trees—it could be the opposite and not really change anything, but this matches our usual intuitions of ordering, as we do with, say, number lines. The point is, you want to be able to know where to look in the tree for values greater or less than the current one you're looking at.

Our `insert` function will insert a value into a tree or, if no tree exists yet, give us a means of building a tree by inserting values. It's important to remember that data is immutable in Haskell. We do not insert a value into an existing tree. Each time we want to insert a value into the data structure, we build a whole new tree:

```
insert' :: Ord a
        => a
        -> BinaryTree a
        -> BinaryTree a
insert' b Leaf = Node Leaf b Leaf
insert' b (Node left a right)
  | b == a = Node left a right
  | b < a  = Node (insert' b left) a right
  | b > a  = Node left a (insert' b right)
```

The base case in our `insert'` function serves a couple purposes. It handles inserting into an empty tree (`Leaf`) and beginning the construction of a new tree and also the case of having reached the bottom of a much larger tree. The simplicity here lets us ignore any inessential differences between those two cases:

```
Prelude> t1 = insert' 0 Leaf
Prelude> t1
Node Leaf 0 Leaf

Prelude> t2 = insert' 3 t1
Prelude> t2
Node Leaf 0 (Node Leaf 3 Leaf)

Prelude> t3 = insert' 5 t2
Prelude> t3
Node Leaf 0
  (Node Leaf 3
```

```
    (Node Leaf 5 Leaf))
```

We will examine binary trees and their properties later in the book. For now, we want to focus not on the properties of binary trees themselves, but on the structure of their type. You might find the following exercises tricky or tedious, but they will deepen your intuition for how recursive types work.

Write map for BinaryTree

Given the definition of `BinaryTree` above, write a map function for the data structure. You don't really need to know anything about binary trees to write these functions. The structure inherent in the definition of the type is all you need. All you need to do is write the recursive functions.

No special algorithms are needed, and we don't expect you to keep the tree balanced or ordered. Also, remember that we've never once *mutated* anything. We've only built new values from input data. Given that, when you go to implement `mapTree`, you're not changing an existing tree—you're building a new one based on an existing one (as when you are mapping functions over lists).

Note, you do *not* need to use `insert'` for this. Retain the original structure of the tree:

```
mapTree :: (a -> b)
        -> BinaryTree a
        -> BinaryTree b
mapTree _ Leaf = Leaf
mapTree f (Node left a right) =
  Node undefined undefined undefined

testTree' :: BinaryTree Integer
testTree' =
  Node (Node Leaf 3 Leaf)
       1
       (Node Leaf 4 Leaf)
```

```
mapExpected =
  Node (Node Leaf 4 Leaf)
       2
       (Node Leaf 5 Leaf)

-- acceptance test for mapTree
mapOkay =
  if mapTree (+1) testTree' == mapExpected
  then print "yup OK!"
  else error "test failed!"
```

Some hints for implementing `mapTree` follow.

The first pattern match in our `mapTree` function is the base case, where we have a `Leaf` value. We can't apply the `f` there, because we don't have an `a`, so we ignore it. Since we have to return a value of type `BinaryTree b` whatever happens, we return a `Leaf` value.

We return a `Node` in the second pattern match of our `mapTree` function. Note that the `Node` data constructor takes three arguments:

```
Prelude> :t Node
Node :: BinaryTree a
     -> a
     -> BinaryTree a
     -> BinaryTree a
```

So, you need to pass it more `BinaryTree`, a single value, and more `BinaryTree`. You have the following terms available to you:

1. `f :: (a -> b)`
2. `left :: BinaryTree a`
3. `a :: a`
4. `right :: BinaryTree a`
5.
```
mapTree :: (a -> b)
           -> BinaryTree a
           -> BinaryTree b
```

The `Node` returned needs to have a value of type `b` and `BinaryTree` values with type `a` inside them. You have two functions at your disposal. One gets you `(a -> b)`, the other maps `BinaryTrees` of type `a` into `BinaryTrees` of type `b`. Get 'em tiger.

A few suggestions that might help you with this exercise:

1. Split out the patterns your function should match on first.
2. Implement the base case first.
3. Try manually writing out the steps of recursion first, then collapse them into a single step that is recursive.

Convert binary trees to lists

Write functions to convert `BinaryTree` values to lists. Make certain your implementation passes the tests:

```
preorder :: BinaryTree a -> [a]
preorder = undefined

inorder :: BinaryTree a -> [a]
inorder = undefined

postorder :: BinaryTree a -> [a]
postorder = undefined

testTree :: BinaryTree Integer
testTree =
  Node (Node Leaf 1 Leaf)
       2
       (Node Leaf 3 Leaf)

testPreorder :: IO ()
testPreorder =
  if preorder testTree == [2, 1, 3]
  then putStrLn "Preorder fine!"
  else putStrLn "Bad news bears."
```

```
testInorder :: IO ()
testInorder =
  if inorder testTree == [1, 2, 3]
  then putStrLn "Inorder fine!"
  else putStrLn "Bad news bears."

testPostorder :: IO ()
testPostorder =
  if postorder testTree == [1, 3, 2]
  then putStrLn "Postorder fine!"
  else putStrLn "Bad news bears"

main :: IO ()
main = do
  testPreorder
  testInorder
  testPostorder
```

Write foldr for BinaryTree

Given the definition of `BinaryTree` we have provided, write a catamorphism for binary trees:

```
-- any traversal order is fine
foldTree :: (a -> b -> b)
         -> b
         -> BinaryTree a
         -> b
```

11.18 Chapter exercises

Multiple choice

1. Given the following datatype:

```
data Weekday =
    Monday
  | Tuesday
  | Wednesday
  | Thursday
  | Friday
```

Which of the following is true?

a) `Weekday` is a type with five data constructors.

b) `Weekday` is a tree with five branches.

c) `Weekday` is a product type.

d) `Weekday` takes five arguments.

2. With the same datatype definition in mind, what is the type of the following function, `f`?

```
f Friday = "Miller Time"
```

a) `f :: [Char]`

b) `f :: String -> String`

c) `f :: Weekday -> String`

d) `f :: Day -> Beer`

3. Types defined with the `data` keyword:

a) Must have at least one argument.

b) Must begin with a capital letter.

c) Must be polymorphic.

d) Cannot be imported from modules.

4. The function `g xs = xs !! (length xs - 1)`:

a) Is recursive and may not terminate.

b) Returns the head of `xs`.

c) Returns the final element of `xs`.

d) Has the same type as `xs`.

Ciphers

In Chapter 9, on lists, you wrote a Caesar cipher. Now, we want to expand on that idea by writing a Vigenère cipher. A Vigenère cipher is another substitution cipher, based on a Caesar cipher, but it uses a series of Caesar ciphers for polyalphabetic substitution. The substitution for each letter in the plain text is determined by a fixed keyword.

So, for example, if you want to encode the message "meet at dawn," the first step is to pick a keyword that will determine which Caesar cipher to use. We'll use the keyword "ALLY" here. You repeat the keyword for as many characters as there are in your original message:

```
MEET AT DAWN
ALLY AL LYAL
```

The number of rightward shifts to make to encode each character is set by the character of the keyword that lines up with it. The "A" means a shift of 0, so the initial M will remain M. But the "L" for our second character sets a rightward shift of 11, so "E" becomes "P." And so on, therefore "meet at dawn" encoded with the keyword "ALLY" becomes "MPPR AE OYWY."

Like the Caesar cipher, you can find all kinds of resources to help you understand this cipher, too, and also many examples written in Haskell. Consider using a combination of `chr`, `ord`, and `mod` again, possibly very similar to what you used for writing the original Caesar cipher.

As-patterns

As-patterns in Haskell are a nifty way to be able to pattern match on part of something and still refer to the entire original value. Some examples:

```
f :: Show a => (a, b) -> IO (a, b)
f t@(a, _) = do
  print a
  return t
```

Here, we pattern match on a tuple so we can get at the first value for printing, but use the `@` symbol to introduce a binding named `t` in order to refer to the whole tuple rather than just a part:

```
Prelude> f (1, 2)
1
(1,2)
```

We can use as-patterns with pattern matching on arbitrary data constructors, which includes lists:

```
doubleUp :: [a] -> [a]
doubleUp [] = []
doubleUp xs@(x:_) = x : xs
```

```
Prelude> doubleUp []
[]
Prelude> doubleUp [1]
[1,1]
Prelude> doubleUp [1, 2]
[1,1,2]
Prelude> doubleUp [1, 2, 3]
[1,1,2,3]
```

Use as-patterns to implement the following functions:

1. This should return `True` if (and only if) all the values in the first list appear in the second list, though they need not be contiguous:

   ```
   isSubseqOf :: (Eq a)
              => [a]
              -> [a]
              -> Bool
   ```

   The following are examples of how this function should work:

   ```
   Prelude> isSubseqOf "blah" "blahwoot"
   True
   Prelude> isSubseqOf "blah" "wootblah"
   True
   ```

```
Prelude> isSubseqOf "blah" "wboloath"
True
Prelude> isSubseqOf "blah" "wootbla"
False
Prelude> isSubseqOf "blah" "halbwoot"
False
Prelude> isSubseqOf "blah" "blawhoot"
True
```

Remember that the sub-sequence has to be in the original order!

2. Split a sentence into words, then tuple each one with its capitalized form:

```
capitalizeWords :: String
                -> [(String, String)]

Prelude> capitalizeWords "hello world"
[("hello", "Hello"), ("world", "World")]
```

Language exercises

1. Write a function that capitalizes a word:

```
capitalizeWord :: String -> String
capitalizeWord = undefined
```

Example output:

```
Prelude> capitalizeWord "Chortle"
"Chortle"
Prelude> capitalizeWord "chortle"
"Chortle"
```

2. Write a function that capitalizes sentences in a paragraph. Recognize when a new sentence has begun by checking for periods. Reuse the `capitalizeWord` function:

```
capitalizeParagraph :: String -> String
capitalizeParagraph = undefined
```

Example result you should get from your function:

```
Prelude> s = "blah. woot ha."
Prelude> capitalizeParagraph s
"Blah. Woot ha."
```

Phone exercise

This exercise by Twitter user @geophf[6] was originally for 1HaskellA-Day.[7] Thank you for letting us use this exercise!

Remember old-fashioned phone inputs for writing text, where you had to press a button multiple times to get different letters to come up? You may still have to do this when you try to search for a movie to watch using your television remote control. You're going to write code to translate sequences of button presses into strings and vice versa.

So! Here is the layout of the phone:

```
-----------------------------------------
|   1       |   2 ABC     |  3 DEF      |
-----------------------------------------
|   4 GHI   |   5 JKL     |  6 MNO      |
-----------------------------------------
|   7 PQRS  |   8 TUV     |  9 WXYZ     |
-----------------------------------------
|   * ^     |   0 + _     |  # .,       |
-----------------------------------------
```

The star (*) capitalizes the current letter, and 0 is your space bar. To represent the digit itself, you press that digit once more than the letters it represents. If you press a button one more than is required to type the digit, it wraps around to the first letter. For example:

```
2     -> 'A'
22    -> 'B'
222   -> 'C'
2222  -> '2'
22222 -> 'A'
```

[6] https://twitter.com/geophf

[7] https://twitter.com/1haskelladay

So on and so forth. We're going to kick this around.

1. Create a data structure that captures the phone layout above. The data structure should be able to express enough of how the layout works that you can use it to dictate the behavior of the functions in the following exercises:

```
-- fill in the rest
data DaPhone = DaPhone
```

2. Convert the following conversations into the key presses required to express them. We're going to suggest types and functions to complete, in order to accomplish the goal, but they're not obligatory. If you want to do it differently, go right ahead:

```
convo :: [String]
convo =
  ["Wanna play 20 questions",
   "Ya",
   "U 1st haha",
   "Lol OK. Have u ever tasted alcohol",
   "Lol ya",
   "Wow ur cool haha. Ur turn",
   "OK. Do u think I am pretty Lol",
   "Lol ya",
   "Just making sure rofl ur turn"]

-- validButtons = "1234567890*#"
type Digit = Char

-- Valid presses: 1 and up
type Presses = Int

reverseTaps :: DaPhone
            -> Char
            -> [(Digit, Presses)]
reverseTaps = undefined
-- assuming the default phone definition
-- 'a' -> [('2', 1)]
-- 'A' -> [('*', 1), ('2', 1)]
```

```
cellPhonesDead :: DaPhone
               -> String
               -> [(Digit, Presses)]
cellPhonesDead = undefined
```

3. How many times do digits need to be pressed for each message?

```
fingerTaps :: [(Digit, Presses)] -> Presses
fingerTaps = undefined
```

4. What is the most popular letter for each message? What was its cost? You'll want to combine `reverseTaps` and `fingerTaps` to figure out what it costs in taps. `reverseTaps` is a list, because you need to press a different button in order to get capitals.

```
mostPopularLetter :: String -> Char
mostPopularLetter = undefined
```

5. What is the most popular letter overall? What is the overall most popular word?

```
coolestLtr :: [String] -> Char
coolestLtr = undefined

coolestWord :: [String] -> String
coolestWord = undefined
```

Hutton's Razor

Hutton's Razor[8] is a simple expression language that expresses integer literals and the addition of values. The "trick" to it is that it's recursive, and the two expressions you're summing together could be literals or themselves further addition operations. This sort of datatype is stereotypical of expression languages used to motivate ideas in research papers and functional pearls. Evaluating or folding a datatype is also in some sense what you're doing most of the time while programming anyway.

1. Your first task is to write the "eval" function that reduces an expression to a final sum:

[8] http://www.cs.nott.ac.uk/~pszgmh/bib.html#semantics

```
data Expr
  = Lit Integer
  | Add Expr Expr

eval :: Expr -> Integer
eval = error "do it to it"
```

Example of expected output:

```
Prelude> eval (Add (Lit 1) (Lit 9001))
9002
```

2. Write a printer for the expressions:

```
printExpr :: Expr -> String
printExpr = undefined
```

Expected output:

```
Prelude> printExpr (Add (Lit 1) (Lit 9001))
"1 + 9001"
Prelude> a1 = Add (Lit 9001) (Lit 1)
Prelude> a2 = Add a1 (Lit 20001)
Prelude> a3 = Add (Lit 1) a2
Prelude> printExpr a3
"1 + 9001 + 1 + 20001"
```

11.19 Definitions

1. A *datatype* is how we declare and create data for our functions to receive as inputs. Datatype declarations begin with the keyword `data`. A datatype is made up of a type constructor and zero or more data constructors, which each have zero or more arguments.

Chapter 12

Signaling Adversity

> Thank goodness we don't
> have only serious problems,
> but ridiculous ones as well.
>
> Edsger W. Dijkstra

12.1 Signaling adversity

Sometimes, it's not convenient or possible for every value in a datatype to make sense for your programs. When that happens in Haskell, we use explicit datatypes to signal when our functions receive a combination of inputs that don't make sense. Later, we'll see how to defend against those adverse inputs at the time we construct our datatypes, but the `Maybe` and `Either` datatypes we will demonstrate here are common.

This chapter will include:

- `Nothing`, or `Just Maybe`.
- `Either` left or right, but not both.
- Higher-kindedness.
- Anamorphisms, but not animorphs.

12.2 How to stop worrying and love Nothing

Let's consider the definition of `Maybe` again:

```
data Maybe a = Nothing | Just a
```

You don't need to define this yourself, as it's included in the `Prelude` by default. It's also a very common datatype in Haskell, because it lets us return a default `Nothing` value when we don't have any sensible values to return for our intended type `a`.

In the following intentionally simplistic function, we could do several things with the odd numbers—we could return them unmodified, we could modify them in some way different from the evens, we could return a zero, or we could write an explicit signal that nothing happens, because the number isn't even:

```
ifEvenAdd2 :: Integer -> Integer
ifEvenAdd2 n =
  if even n then n + 2 else ???
```

What can we do to make it say, "Hey, this number isn't even, so I have nothing for you, my friend?" Instead of promising an `Integer` result, we can return `Maybe Integer`:

```
ifEvenAdd2 :: Integer -> Maybe Integer
ifEvenAdd2 n =
  if even n then n + 2 else Nothing
```

This isn't quite complete or correct, either. While `Nothing` has the type `Maybe a`, and `a` can be assumed to be any type the `Maybe` constructor could contain, `n + 2` is still of type `Integer`. We need to wrap that value in the other constructor `Maybe` provides: `Just`. Here's the error you'd get if you tried to load it:

```
• Couldn't match expected type
    ‘Maybe Integer’
              with actual type ‘Integer’
• In the expression: n + 2
  In the expression:
    if even n then n + 2 else Nothing
  In an equation for ‘ifEvenAdd2’:
      ifEvenAdd2 n =
        if even n then n + 2 else Nothing
```

And here's how we fix it:

```
ifEvenAdd2 :: Integer -> Maybe Integer
ifEvenAdd2 n =
  if even n then Just (n+2) else Nothing
```

We have to parenthesize `(n + 2)`, because function application binds the most tightly in Haskell (has the highest precedence), so the compiler would otherwise parse it as `(Just n) + 2`, which is wrong and throws a type error. Now our function is correct and explicit about the possibility of not getting a result!

Smart constructors for datatypes

Let's consider a `Person` type that keeps track of two things, a person's name and their age. We'll write this up as a simple product type (note that `Name` and `Age` are type aliases):

```
type Name = String
type Age = Integer

data Person = Person Name Age deriving Show
```

There are already a few problems here. One is that we could construct a `Person` with an empty `String` for a name or make a person who is negative years old. This is no problem to fix with `Maybe`, though:

```
type Name = String
type Age = Integer

data Person = Person Name Age deriving Show

mkPerson :: Name -> Age -> Maybe Person
mkPerson name age
  | name /= "" && age >= 0 =
      Just $ Person name age
  | otherwise = Nothing
```

And if you load this into your REPL:

```
Prelude> mkPerson "John Browning" 160
Just (Person "John Browning" 160)
```

Cool. What happens when we feed it adverse data?

```
Prelude> mkPerson "" 160
Nothing
Prelude> mkPerson "blah" 0
Just (Person "blah" 0)
Prelude> mkPerson "blah" (-9001)
Nothing
```

`mkPerson` is what we call a *smart constructor*. It allows us to construct values of a type only when they meet certain criteria, so that we know we have a valid value, and return an explicit signal when they do not.

This is much better than our original version, but what if we want to know whether it is the name, age, or both that are invalid? We may want to tell our user something is wrong with their input. Fortunately, we have a datatype for that!

12.3 Bleating either

We want a way to express why we don't get a successful result back from our `mkPerson` constructor. To handle that, we've got the `Either` datatype, which is defined as follows in the `Prelude`:

```
data Either a b = Left a | Right b
```

What we want is a way to know *why* our inputs are incorrect, *if* they are incorrect. So, we'll start by making a sum type to enumerate our failure modes:

```
data PersonInvalid = NameEmpty
                   | AgeTooLow
                   deriving (Eq, Show)
```

By now, you know why we derive `Show`, but it's important that we derive `Eq`, as well, because otherwise we can't equality check the constructors. Pattern matching is a case expression, where the data constructor is the condition. Case expressions and pattern matching *will work* without an `Eq` instance, but guards using `==` will not. As we've shown you previously, you can write your own `Eq` instance for your datatype if you want a specific behavior, but it's usually not necessary to do, so we will usually derive the `Eq` instance. Here's the difference demonstrated in code:

```
module EqCaseGuard where

data PersonInvalid = NameEmpty
                   | AgeTooLow
```

This compiles without an `Eq` instance:

```
toString :: PersonInvalid -> String
toString NameEmpty = "NameEmpty"
toString AgeTooLow = "AgeTooLow"

instance Show PersonInvalid where
  show = toString
```

This doesn't work without an `Eq` instance:

```
blah :: PersonInvalid -> String
blah pi
  | pi == NameEmpty = "NameEmpty"
  | pi == AgeTooLow = "AgeTooLow"
  | otherwise = "???"
```

It's worth considering that if you need to have an `Eq` instance to pattern match, how would you write `Eq` instances for your datatypes?

Next, our constructor type is going to change to:

```
mkPerson :: Name
         -> Age
         -> Either PersonInvalid Person
```

This signifies that we're going to get a `Person` value if we succeed but a `PersonInvalid` if it fails. Now, we need to change our logic to return `PersonInvalid` values inside a `Left` constructor when the data is invalid, discriminating by each case as we go:

```
type Name = String
type Age = Integer

data Person = Person Name Age deriving Show

data PersonInvalid = NameEmpty
                   | AgeTooLow
                   deriving (Eq, Show)

mkPerson :: Name
         -> Age
         -> Either PersonInvalid Person
--           [1]       [2]        [3]
mkPerson name age
  | name /= "" && age >= 0 =
      Right $ Person name age
--            [4]
  | name == "" = Left NameEmpty
--                    [5]
  | otherwise = Left AgeTooLow
```

1. Our `mkPerson` type takes a `Name` and `Age` and returns an `Either` result.
2. The `Left` result of the `Either` is an invalid person, when either the name or age is an invalid input.
3. The `Right` result is a valid `Person`.
4. The first case of our `mkPerson` function, then, matches on the `Right` constructor of the `Either` and returns a `Person` result. We could have written:

   ```
   name /= "" && age >= 0 =
     Right (Person name age)
   ```

   Instead of using the dollar sign operator.
5. The next two cases match on the `Left` constructor and allow us to tailor our invalid results based on the failure reasons. We can pattern match on `Left`, because it's one of the constructors of `Either`.

We use `Left` as our invalid or error constructor for a couple of reasons. It is conventional to do so in Haskell, but that convention came about for a reason. The reason has to do with the ordering of type arguments and application of functions. Normally, it is your error or invalid result that is going to cause to stop whatever work is being done by your program. `Functor` will not map over the left type argument, because it is applied away. You may remember `Functor` from our introduction of `fmap` back in Chapter 9, on lists; don't worry, a full explanation of `Functor` is coming soon. Since you normally want to apply functions and map over the case that *doesn't* stop your program (that is, *not* the error case), it has become convention that the `Left` of `Either` is used for whatever case is going to cause the work to stop.

Let's see what it looks like when we have good data, although Djali isn't a person:[1]

```
Prelude> :t mkPerson "Djali" 5
mkPerson "Djali" 5 ::
```

[1] Don't know what we mean? Check the name Djali on a search engine.

```
  Either PersonInvalid Person
Prelude> mkPerson "Djali" 5
Right (Person "Djali" 5)
```

Then, we can see what this does for us when dealing with bad data:

```
Prelude> mkPerson "" 10
Left NameEmpty
Prelude> mkPerson "Djali" (-1)
Left AgeTooLow
Prelude> mkPerson "" (-1)
Left NameEmpty
```

Notice in the last example that when both the name and the age are wrong, we're only going to see the result of the first failure case, not both.

This is imperfect in one respect, as it doesn't let us express a list of errors. We can fix this, too! One thing that will change is that instead of validating all the data for a `Person` at once, we're going to make separate checking functions and then combine the results. We'll see means of abstracting patterns like this out later. We're adding a type alias that isn't in our previous version. Otherwise, these types are the same as above:

```
type Name = String
type Age = Integer

type ValidatePerson a =
  Either [PersonInvalid] a

data Person = Person Name Age deriving Show

data PersonInvalid = NameEmpty
                   | AgeTooLow
                   deriving (Eq, Show)
```

Now, we'll write our checking functions. Although more than one thing could hypothetically be wrong with the age value, we'll keep this simple and only check to make sure it's a positive `Integer` value:

```
ageOkay :: Age
        -> Either [PersonInvalid] Age
ageOkay age = case age >= 0 of
  True  -> Right age
  False -> Left [AgeTooLow]

nameOkay :: Name
         -> Either [PersonInvalid] Name
nameOkay name = case name /= "" of
  True  -> Right name
  False -> Left [NameEmpty]
```

We can nest the `PersonInvalid` sum type right into the `Left` position of `Either`, just as we saw in the previous chapter (although we weren't using `Either` there but similar types).

A couple of things to note here:

- The `Name` value will only return this invalid result when it's an empty `String`.
- Since `Name` is only a `String` value, it can be any `String` with characters inside it, so `"42"` will be returned as a valid name. Try it yourself.
- If you try to substitute an `Integer` for the name, you won't get a `Left` result, you'll get a type error. Try it. You'll get a similar result if you try to feed a string value to the `ageOkay` function.
- We're going to return a list of `PersonInvalid` results. That will allow us to return *both* `NameEmpty` and `AgeTooLow` in cases where both of those are true.

Now that our functions rely on `Either` to validate that the age and name values are independently valid, we can write a `mkPerson` function that will use our type alias `ValidatePerson`:

```
mkPerson :: Name
         -> Age
         -> ValidatePerson Person
--              [1]          [2]
```

```
mkPerson name age =
  mkPerson' (nameOkay name) (ageOkay age)
--  [3]          [4]           [5]

mkPerson' :: ValidatePerson Name
          -> ValidatePerson Age
          -> ValidatePerson Person
--                [6]

mkPerson' (Right nameOk) (Right ageOk) =
  Right (Person nameOk ageOk)
mkPerson' (Left badName) (Left badAge) =
  Left (badName ++ badAge)
mkPerson' (Left badName) _ = Left badName
mkPerson' _ (Left badAge) = Left badAge
```

1. A type alias for `Either [PersonInvalid] a`.
2. This is the `a` argument to the `ValidatePerson` type.
3. Our main function now relies on a similarly-named helper function.
4. The first argument to this function is the result of the `nameOkay` function.
5. The second argument is the result of the `ageOkay` function.
6. The type relies on the synonym for `Either`.

The rest of our helper function `mkPerson'` consists of plain old pattern matches.

Let's see what we get:

```
Prelude> mkPerson "" (-1)
Left [NameEmpty,AgeTooLow]
```

Ahh, that's more like it. This time, we can tell the user what is incorrect in one go without them having to round-trip each mistake! Later in the book, we'll be able to replace `mkPerson` and `mkPerson'` with the following:

```
mkPerson
  :: Name
  -> Age
  -> Validation [PersonInvalid] Person

mkPerson name age =
  liftA2
    Person (nameOkay name) (ageOkay age)
```

12.4 Kinds, a thousand stars in your types

Kinds are types one level up. They are used to describe the types of type constructors. One noteworthy feature of Haskell is that it has *higher-kinded types*. The term "higher-kinded" derives from higher-order functions, functions that take more functions as arguments. Type constructors (that is, higher-kinded types) are types that take more types as arguments. The Haskell Report uses the term *type constant* to refer to types that take no arguments and are already concrete types. In the Report, *type constructor* is used to refer to a type that must have arguments applied in order to become a concrete type.

As we discussed in the last chapter, these are examples of *type constants*:

```
Prelude> :kind Int
Int :: *
Prelude> :k Bool
Bool :: *
Prelude> :k Char
Char :: *
```

The `::` syntax usually means "has the type of," but it is used for kind signatures as well as type signatures.

The following is an example of a type that has a type *constructor* rather than a type *constant*:

```
data Example a = Blah | Woot a
```

`Example` is a type constructor rather than a constant, because it takes a type argument `a` that is used with the `Woot` data constructor. In GHCi, we can query kinds with the `:k` command:

```
data Example a = Blah | Woot a
```

```
Prelude> :k Example
Example :: * -> *
```

`Example` has one parameter, so it must be applied to one type in order to become a concrete type represented by a single `*`. The 2-tuple takes two arguments, so it must be applied to two types to become a concrete type:

```
Prelude> :k (,)
(,) :: * -> * -> *


Prelude> :k (Int, Int)
(Int, Int) :: *
```

The `Maybe` and `Either` datatypes we've just reviewed also have type constructors rather than constants. They have to be applied to an argument before they become concrete types. As with the effect of currying in type signatures, applying `Maybe` to an `a` type constructor relieves us of one arrow and gives it the kind `*`, or star:

```
Prelude> :k Maybe
Maybe :: * -> *
Prelude> :k Maybe Int
Maybe Int :: *
```

On the other hand, `Either` has to be applied to two arguments, an `a` and a `b`, so the kind of `Either` is "star to star to star":

```
Prelude> :k Either
Either :: * -> * -> *
```

And, again, we can query the effects of applying it to arguments:

```
Prelude> :k Either Int
Either Int :: * -> *
Prelude> :k Either Int String
Either Int String :: *
```

As we've said, the kind `*` represents a concrete type. There is nothing left awaiting application.

Lifted and unlifted types To be precise, kind `*` is the kind of all standard lifted types, while types that have the kind `#` are unlifted. A lifted type, which includes any datatype you could define yourself, is any that can be inhabited by *bottom*. Lifted types are represented by a pointer and include most of the datatypes we've seen and most that you're likely to encounter and use. Unlifted types are any types that *cannot* be inhabited by bottom. Types of kind `#` are often native machine types and raw pointers. Newtypes are a special case in that they are kind `*`, but they are unlifted, because their representation is identical to that of the type they contain, so a newtype itself is not creating any new pointer beyond that of the type it contains. That fact means that the newtype itself cannot be inhabited by bottom—only the thing it contains can be—so newtypes are unlifted. The default kind of concrete, fully-applied datatypes in GHC is kind `*`.

Now, what happens if we let our type constructor take an argument?

```
data Identity a = Identity a
```

```
Prelude> :k Identity
Identity :: * -> *
```

As we discussed in the previous chapter, the arrow in the kind signature, like the function arrow in type signatures, signals a need for application. In this case, we construct the type by applying it to another type.

Let's consider the case of `Maybe`, which is defined as follows:

```
data Maybe a = Nothing | Just a
```

The type `Maybe` is a type constructor, because it takes one argument before it becomes a concrete type:

```
Prelude> :k Maybe
Maybe :: * -> *

Prelude> :k Maybe Int
```

```
Maybe Int :: *
Prelude> :k Maybe Bool
Maybe Bool :: *

Prelude> :k Int
Int :: *
Prelude> :k Bool
Bool :: *
```

Whereas the following will not work, because the kinds don't match up:

```
Prelude> :k Maybe Maybe

• Expecting one more argument to ‘Maybe’
  Expected a type, but ‘Maybe’ has kind
    ‘* -> *’
• In the first argument of ‘Maybe’,
    namely ‘Maybe’
  In the type ‘Maybe Maybe’
```

`Maybe` expects a single type argument of kind `*`, which `Maybe` is not.

If we give `Maybe` a type argument that is kind `*`, it also becomes kind `*`, so then it can be an argument to another `Maybe`:

```
Prelude> :k Maybe Char
Maybe Char :: *

Prelude> :k Maybe (Maybe Char)
Maybe (Maybe Char) :: *
```

Our `Example` datatype from earlier also won't work as an argument for `Maybe`, by itself:

```
data Example a = Blah | Woot a

Prelude> :k Maybe Example

• Expecting one more argument to ‘Example’
  Expected a type, but ‘Example’ has kind
```

```
    ‘* -> *’
• In the first argument of ‘Maybe’,
    namely ‘Example’
  In the type ‘Maybe Example’
```

However, if we apply the `Example` type constructor, we can make it work and create a value of that type:

```
Prelude> :k Maybe (Example Int)
Maybe (Example Int) :: *
Prelude> :t Just (Woot n)
Just (Woot n) :: Maybe (Example Int)
```

Note that the list type constructor `[]` is also kind `* -> *` and otherwise unexceptional save for the bracket syntax that lets you type `[a]` and `[Int]` instead of `[] a` and `[] Int`:

```
Prelude> :k []
[] :: * -> *

Prelude :k [] Int
[] Int :: *

Prelude> :k [Int]
[Int] :: *
```

So, we can't have a `Maybe []` for the same reason we couldn't have a `Maybe Maybe`, but we can have a `Maybe [Bool]`:

```
Prelude> :k Maybe []

• Expecting one more argument to ‘[]’
  Expected a type, but ‘[]’ has kind
    ‘* -> *’
• In the first argument of ‘Maybe’,
    namely ‘[]’
  In the type ‘Maybe []’

Prelude> :k Maybe [Bool]
Maybe [Bool] :: *
```

If you recall, one of the first times we used `Maybe` in this book was to write a safe version of a `tail` function:

```
safeTail        :: [a] -> Maybe [a]
safeTail []     = Nothing
safeTail (_:[]) = Nothing
safeTail (_:xs) = Just xs
```

As soon as we apply this to a value, the polymorphic type variables become constrained or concrete types:

```
Prelude> safeTail "julie"
Just "ulie"
Prelude> :t safeTail "julie"
safeTail "julie" :: Maybe [Char]

Prelude> safeTail [1..10]
Just [2,3,4,5,6,7,8,9,10]
Prelude> :t safeTail [1..10]
safeTail [1..10]
  :: (Num a, Enum a) => Maybe [a]

Prelude> :t safeTail [1..10 :: Int]
safeTail [1..10 :: Int] :: Maybe [Int]
```

We can expand on type constructors that take a single argument and see how the kind changes as we go:

```
data Trivial = Trivial
data Unary a = Unary a
data TwoArgs a b = TwoArgs a b
data ThreeArgs a b c = ThreeArgs a b c
```

```
Prelude> :k Trivial
Trivial :: *
Prelude> :k Unary
Unary :: * -> *
Prelude> :k TwoArgs
TwoArgs :: * -> * -> *
Prelude> :k ThreeArgs
ThreeArgs :: * -> * -> * -> *
```

It may not be clear why this is useful to know right now, other than that it helps you to understand when your type errors are caused by things not being fully applied. The implications of higher-kindedness will become clearer in a later chapter.

Data constructors are functions

In the previous chapter, we noted the difference between data constants and data constructors and also noted that data constructors that haven't been fully applied have function arrows in them. Once you apply them to their arguments, they return a value of the appropriate type. In other words, data constructors are functions. As it happens, they behave like Haskell functions in that they are curried, as well.

First, let's observe that nullary data constructors, which are values taking no arguments, are *not* like functions:

```
data Trivial = Trivial deriving Show
```

```
Prelude> Trivial 1

• Couldn't match expected type
    ‘Integer -> t’
              with actual type ‘Trivial’
...etc...
```

However, data constructors that take arguments *do* behave like functions:

```
data UnaryC = UnaryC Int deriving Show
```

```
Prelude> :t UnaryC
UnaryC :: Int -> UnaryC
Prelude> UnaryC 10
UnaryC 10
Prelude> :t UnaryC 10
UnaryC 10 :: UnaryC
```

Like functions, their arguments are type checked against the specification in the type:

```
Prelude> UnaryC "blah"

• Couldn't match expected type ‘Int’ with
    actual type ‘[Char]’
• In the first argument of ‘UnaryC’,
    namely ‘"blah"’
  In the expression: UnaryC "blah"
  In an equation for ‘it’: it =
    UnaryC "blah"
```

If we want a unary data constructor that can contain any type, we would parameterize the type, like so:

```
data Unary a = Unary a deriving Show
```

```
Prelude> :t Unary
Unary :: a -> Unary a
Prelude> :t Unary 10
Unary 10 :: Num a => Unary a
Prelude> :t Unary "blah"
Unary "blah" :: Unary [Char]
```

And again, this works just like a function, except the type of the argument can be whatever we want.

Note that if we want to use a derived (GHC generated) `Show` instance for `Unary`, it has to be able to also show the contents, the type `a` value contained by `Unary`'s data constructor:

```
Prelude> :info Unary
data Unary a = Unary a
instance Show a => Show (Unary a)
```

If we try to use a type for `a` that does not have a `Show` instance, it won't cause a problem until we try to show the value:

```
Prelude> :t (Unary id)
(Unary id) :: Unary (a -> a)

Prelude> show (Unary id)
```

```
• No instance for (Show (a0 -> a0)) arising
    from a use of ‘show’
    (maybe you haven't applied a function
      to enough arguments?)
• In the expression: show (Unary id)
  In an equation for ‘it’: it =
    show (Unary id)
```

The only way to avoid this would be to write an instance that does not show the value contained in the `Unary` data constructor, but that would be somewhat unusual.

Another thing to keep in mind is that you can't ordinarily hide polymorphic types from your type constructor, so the following is invalid:

```
data Unary = Unary a deriving Show

error: Not in scope: type variable ‘a’
```

In order for the type variable `a` to be in scope, we usually need to introduce it with our type constructor. There are ways around this, but they're rarely necessary or a good idea and not relevant to the beginning Haskeller.

Here's an example using `fmap` and the `Just` data constructor from `Maybe` to demonstrate how `Just` is also like a function:

```
Prelude> fmap Just [1, 2, 3]
[Just 1,Just 2,Just 3]
```

The significance and utility of this may not be immediately obvious but will be clearer in later chapters.

12.5 Chapter exercises

Determine the kinds

1. Given:

   ```
   id :: a -> a
   ```

   What is the kind of `a`?

2. `r :: a -> f a`

   What are the kinds of `a` and `f`?

String processing

Because this is the kind of thing linguists *ahem* enjoy doing in their spare time.

1. Write a recursive function named `replaceThe` that takes a text/string, breaks it into words, and replaces each instance of `"the"` with `"a"`. It should only replace exactly the word `"the"`. `notThe` is a suggested helper function for accomplishing this:

   ```
   notThe :: String -> Maybe String
   notThe = undefined
   ```

   Example GHCi session using the above functions:

   ```
   Prelude> notThe "the"
   Nothing
   Prelude> notThe "blahtheblah"
   Just "blahtheblah"
   Prelude> notThe "woot"
   Just "woot"

   replaceThe :: String -> String
   replaceThe = undefined

   Prelude> replaceThe "the cow loves us"
   "a cow loves us"
   ```

2. Write a recursive function that takes a text/string, breaks it into words, and counts the number of instances of `"the"` followed by a vowel-initial word:

   ```
   countTheBeforeVowel :: String -> Integer
   countTheBeforeVowel = undefined
   ```

```
Prelude> countTheBeforeVowel "the cow"
0
Prelude> countTheBeforeVowel "the evil cow"
1
```

3. Return the number of letters that are vowels in a word.

   Hint: it's helpful to break this into steps. Add any helper functions necessary to achieve your objectives:

   a) Test for vowel-hood.

   b) Return the vowels of a string.

   c) Count the number of elements returned.

```
countVowels :: String -> Integer
countVowels = undefined

Prelude> countVowels "the cow"
2
Prelude> countVowels "Mikolajczak"
4
```

Validate the word

Use the `Maybe` type to write a function that counts the number of vowels in a string and the number of consonants. If the number of vowels exceeds the number of consonants, the function returns `Nothing`. In many human languages, vowels rarely exceed the number of consonants, so when they do, it *may* indicate the input isn't a word (that is, a valid input to your dataset):

```
newtype Word' =
  Word' String
  deriving (Eq, Show)

vowels = "aeiou"

mkWord :: String -> Maybe Word'
mkWord = undefined
```

It's only natural

You'll be presented with a datatype to represent the natural numbers. The only values representable with the naturals are whole numbers from zero to infinity. Your task will be to implement functions to convert natural numbers to integers and integers to naturals. The conversion from `Nat` to `Integer` won't return `Maybe`, because—as you know—`Integer` is a strict superset of `Nat`. Any `Nat` can be represented by an `Integer`, but the same is *not* true of any `Integer`. Negative numbers are not valid natural numbers:

```
-- As natural as any
-- competitive bodybuilder
data Nat =
    Zero
  | Succ Nat
  deriving (Eq, Show)

natToInteger :: Nat -> Integer
natToInteger = undefined

integerToNat :: Integer -> Maybe Nat
integerToNat = undefined
```

```
Prelude> natToInteger Zero
0
Prelude> natToInteger (Succ Zero)
1
Prelude> natToInteger (Succ (Succ Zero))
2

Prelude> integerToNat 0
Just Zero
Prelude> integerToNat 1
Just (Succ Zero)
Prelude> integerToNat 2
Just (Succ (Succ Zero))
Prelude> integerToNat (-1)
Nothing
```

Small library for Maybe

Write the following functions. This may take some time.

1. Simple Boolean checks for `Maybe` values:

```
isJust :: Maybe a -> Bool
isNothing :: Maybe a -> Bool

Prelude> isJust (Just 1)
True
Prelude> isJust Nothing
False
Prelude> isNothing (Just 1)
False
Prelude> isNothing Nothing
True
```

2. The following is the `Maybe` catamorphism. You can turn a `Maybe` value into anything else with this:

```
mayybee :: b -> (a -> b) -> Maybe a -> b

Prelude> mayybee 0 (+1) Nothing
0
Prelude> mayybee 0 (+1) (Just 1)
2
```

3. In case you just want to provide a fallback value. Try writing it in terms of the `maybe` catamorphism:

```
fromMaybe :: a -> Maybe a -> a

Prelude> fromMaybe 0 Nothing
0
Prelude> fromMaybe 0 (Just 1)
1
```

4. Converting between `List` and `Maybe`:

```
listToMaybe :: [a] -> Maybe a
maybeToList :: Maybe a -> [a]

Prelude> listToMaybe [1, 2, 3]
Just 1
Prelude> listToMaybe []
Nothing
Prelude> maybeToList (Just 1)
[1]
Prelude> maybeToList Nothing
[]
```

5. For when we want to drop the `Nothing` values from a list:

```
catMaybes :: [Maybe a] -> [a]

Prelude> catMaybes [Just 1, Nothing, Just 2]
[1, 2]
Prelude> let xs = take 3 $ repeat Nothing
Prelude> catMaybes xs
[]
```

6. You'll see this called `sequence` later:

```
flipMaybe :: [Maybe a] -> Maybe [a]

Prelude> flipMaybe [Just 1, Just 2, Just 3]
Just [1, 2, 3]
Prelude> flipMaybe [Just 1, Nothing, Just 3]
Nothing
```

Small library for Either

Write each of the following functions. If more than one possible unique function exists for the type, use common sense to determine what it should do.

1. Try to eventually arrive at a solution that uses `foldr`, even if earlier versions don't use `foldr`:

```
lefts' :: [Either a b] -> [a]
```

2. Same as the last one. Use `foldr`, eventually:

```
rights' :: [Either a b] -> [b]
```

3. ```
   partitionEithers' :: [Either a b]
                     -> ([a], [b])
   ```

4. ```
   eitherMaybe' :: (b -> c)
                -> Either a b
                -> Maybe c
   ```

5. This is a general catamorphism for `Either` values:

```
either' :: (a -> c)
        -> (b -> c)
        -> Either a b
        -> c
```

6. Same as before, but use the `either'` function you just wrote:

```
eitherMaybe'' :: (b -> c)
              -> Either a b
              -> Maybe c
```

Most of the functions you just saw are in the `Prelude`, `Data.Maybe`, or `Data.Either`, but you should strive to write them yourself without looking at existing implementations. You will deprive *yourself* if you cheat.

Unfolds

While the idea of catamorphisms is still relatively fresh in our minds, let's turn our attention to their dual: *anamorphisms*. If folds, or catamorphisms, let us break data structures down, then unfolds let us build them up. There are, as with folds, a few different ways to unfold a data structure. We can use them to create finite and infinite data structures alike.

`iterate` is like a limited unfold that never ends:

```
Prelude> :t iterate
iterate :: (a -> a) -> a -> [a]
```

`iterate` never ends, so we must use `take` to get a finite list:

```
Prelude> take 10 $ iterate (+1) 0
[0,1,2,3,4,5,6,7,8,9]
```

`unfoldr` is more general than `iterate`:

```
Prelude> :t unfoldr
unfoldr :: (b -> Maybe (a, b)) -> b -> [a]
```

Here's how to use `unfoldr` to do what `iterate` does:

```
Prelude> f b = Just (b, b + 1)
Prelude> take 10 $ unfoldr f 0
[0,1,2,3,4,5,6,7,8,9]
```

Why bother?

We bother with this for the same reason we abstract direct recursion into folds, such as with `sum`, `product`, and `concat`:

```
import Data.List

mehSum :: Num a => [a] -> a
mehSum xs = go 0 xs
  where go :: Num a => a -> [a] -> a
        go n [] = n
        go n (x:xs) = (go (n + x) xs)

niceSum :: Num a => [a] -> a
niceSum = foldl' (+) 0

mehProduct :: Num a => [a] -> a
mehProduct xs = go 1 xs
  where go :: Num a => a -> [a] -> a
        go n [] = n
        go n (x:xs) = (go (n * x) xs)
```

```
niceProduct :: Num a => [a] -> a
niceProduct = foldl' (*) 1
```

Remember the redundant structure when we looked at folds?

```
mehConcat :: [[a]] -> [a]
mehConcat xs = go [] xs
  where go :: [a] -> [[a]] -> [a]
        go xs' [] = xs'
        go xs' (x:xs) = (go (xs' ++ x) xs)

niceConcat :: [[a]] -> [a]
niceConcat = foldr (++) []
```

This may give you a mild headache, but you may also see that this same principle of abstracting out common patterns and giving them names applies as well to unfolds as it does to folds.

Write your own iterate and unfoldr

1. Write the function `myIterate` using direct recursion. Compare the behavior with the built-in `iterate` to gauge correctness. Do not look at the source or any examples of `iterate`, so that you are forced to do this yourself:

   ```
   myIterate :: (a -> a) -> a -> [a]
   myIterate = undefined
   ```

2. Write the function `myUnfoldr` using direct recursion. Compare with the built-in `unfoldr` to check your implementation. Again, don't look at implementations of `unfoldr`, so that you figure it out yourself:

   ```
   myUnfoldr :: (b -> Maybe (a, b))
             -> b
             -> [a]
   myUnfoldr = undefined
   ```

3. Rewrite `myIterate` into `betterIterate` using `myUnfoldr`. A hint—we use `unfoldr` to produce the same results as `iterate` above. Do this

with different functions, and see if you can abstract the structure out.

It helps to have the types in front of you:

```
myUnfoldr :: (b -> Maybe (a, b))
          -> b
          -> [a]

betterIterate :: (a -> a) -> a -> [a]
betterIterate f x = myUnfoldr ...?
```

Remember, your `betterIterate` should have the same results as `iterate`:

```
Prelude> take 10 $ iterate (+1) 0
[0,1,2,3,4,5,6,7,8,9]

Prelude> take 10 $ betterIterate (+1) 0
[0,1,2,3,4,5,6,7,8,9]
```

Finally something other than a list!

Given the `BinaryTree` from the last chapter, complete the following exercises. Here's that datatype again:

```
data BinaryTree a =
    Leaf
  | Node (BinaryTree a) a (BinaryTree a)
  deriving (Eq, Ord, Show)
```

1. Write `unfold` for `BinaryTree`:

   ```
   unfold :: (a -> Maybe (a,b,a))
          -> a
          -> BinaryTree b
   unfold = undefined
   ```

2. Make a tree builder.

   Using the `unfold` function you made for `BinaryTree`, write the following function:

```
treeBuild :: Integer -> BinaryTree Integer
treeBuild n = undefined
```

You should be producing results that look like the following:

```
Prelude> treeBuild 0
Leaf
Prelude> treeBuild 1
Node Leaf 0 Leaf
Prelude> treeBuild 2
Node (Node Leaf 1 Leaf)
     0
     (Node Leaf 1 Leaf)
Prelude> treeBuild 3
Node (Node (Node Leaf 2 Leaf)
           1
           (Node Leaf 2 Leaf))
     0
     (Node (Node Leaf 2 Leaf)
           1
           (Node Leaf 2 Leaf))
```

Or in a slightly different representation:

```
  0


  0
 / \
 1 1

  0
 / \
 1  1
/\  /\
2 2  2 2
```

Good work.

12.6 Definitions

1. A *higher-kinded type* type is any type whose kind has a function arrow in it and which can be described as a type constructor rather than as a type constant. The following types are of a higher kind than *:

```
Maybe  :: * -> *
[]     :: * -> *
Either :: * -> * -> *
(->)   :: * -> * -> *
```

   The following are not:

```
Int    :: *
Char   :: *
String :: *
[Char] :: *
```

   This is not to be confused with higher-kinded *polymorphism*, which we'll discuss later.

Chapter 13

Building Projects

> Wherever there is modularity there is the potential for misunderstanding: hiding information implies a need to check communication.
>
> Alan Perlis

13.1 Modules

Haskell programs are organized into modules. Modules contain the datatypes, type synonyms, type classes, type class instances, and values you define at the top level. They offer a means to import other modules into the scope of your program, and they also contain values that can be exported to other modules. If you've ever used a language with namespaces, it's the same thing.

In this chapter, we will be building a small, interactive hangman-style game. Students of Haskell often ask what kind of project they should work on as a way to learn Haskell, and they want to jump right into the kind of program they're used to building in the languages they already know. What most often happens is the student realizes how much they still don't understand about Haskell, shakes their fist at the sky, curses Haskell's very name and all the elitist jerks who write Haskell, and flees to relative safety. Nobody wants that. Haskell is sufficiently different from other languages that we think it's best to spend time getting comfortable with how Haskell itself works before trying to build substantial projects.

This chapter's primary focus is not so much on code but on how to set up a project in Haskell, use the package manager known as Cabal, build the project with Stack, and work with Haskell modules as they are. There are a few times we ask you to implement part of the hangman game yourself, but much of the code is already written for you, and we've tried to explain the structure as well as we can at this point in the book. Some of it you won't properly understand until we've covered at least monads and `IO`. But if you finish the chapter feeling like you now know how to set up a project environment and get things running, then this chapter will have accomplished its goal and we'll all go off and take a much needed, mid-book nap.

Try to relax and have fun with this. You've earned it after those binary tree exercises.

In this chapter, we'll cover:

- Writing Haskell programs with modules.
- Using the Cabal package manager.
- Building our project with Stack.

- Conventions around project organization.
- Designing a small interactive game.

Note that you'll need to have Stack[1] and Git[2] to follow along with the instructions in this chapter. We'll be using Git to download an example project. Depending on your level of prior experience, some of this may not be new information for you. Feel free to move as quickly through this material as feels comfortable.

13.2 Making packages with Stack

Haskell Cabal, or Common Architecture for Building Applications and Libraries, is a package manager. A *package* is a program you're building, including all of its modules and dependencies, whether you've written it or you're building someone else's program. A package has *dependencies*, which are the interlinked elements of that program, the other packages and libraries it may depend on, and any tests and documentation associated with the project. Cabal exists to help organize all this and make sure all of your dependencies are properly in scope.

Stack is a cross-platform program for developing Haskell projects. It is aimed at Haskellers both new and experienced, and it helps you manage both projects made up of multiple packages as well as individual packages, whereas Cabal exists primarily to describe a single package with a Cabal file that has the `.cabal` file extension.

Stack is built on top of Cabal in some important senses, so we will still be working with `.cabal` files. However, Stack simplifies the process somewhat, especially in large projects with multiple dependencies, by allowing you to build those large libraries only once and use them across projects. Stack also relies on an LTS (long term support) snapshot of Haskell packages from a central repository called Stackage[3] that are guaranteed to work together, unlike packages from Hackage, another repository, which may have conflicting dependencies.

[1] `https://haskellstack.org`
[2] `https://git-scm.com/`
[3] `https://www.stackage.org/`

While the Haskell community does not have a prescribed project layout, we recommend the basic structure embodied in the Stack templates.

13.3 Working with a basic project

We're going to start learning Cabal and Stack by building a sample project called `hello`. To make this less tedious, we're going to use Git to checkout the sample project. In an appropriate directory for storing your projects, you'll want to `git clone` the repository `https://github.com/haskellbook/hello`.

Building the project

Change into the project directory that the `git clone` invocation creates:

```
$ cd hello
```

You could edit the `hello.cabal` file. There you can replace "Your Name Here" with... your name. Next, we'll build our project:

```
$ stack build
```

If it complains about needing GHC to be installed, don't panic! Part of the benefit of Stack is that it can manage your GHC installs for you. Before re-attempting `stack build`, do the following:

```
$ stack setup
```

The `setup` command for Stack determines which version of GHC you need based on the LTS snapshot specified in the `stack.yaml` file of your project. The `stack.yaml` file is used to determine the versions of your packages and which version of GHC they'll work best with. If you don't need to do this, it's possible you have a compatible version of GHC already installed or that you already ran setup for an LTS snapshot that needed the same version of GHC in the past. To learn more about this, check out the Stackage website.

Loading and running code from the REPL

Having done that, next we'll fire up the REPL:

```
$ stack ghci
...some other noise...
Ok, modules loaded: Main.
Prelude> :l Main
[1 of 1] Compiling Main
Ok, one module loaded.
Prelude> main
hello world
```

Above, we successfully start a GHCi REPL that is aware of our project, load our `Main` module, and then run the `main` function. Using Stack's GHCi integration to fire up a REPL doesn't just let us load and run code in our project, it also enables us to make use of our project's dependencies. We'll demonstrate this later.

stack exec

When you ran `build` earlier, you may have seen something like:

```
Linking .stack-work/dist/...noise.../hello
```

This noise is Stack compiling an executable binary and linking to it. You can type the full path that Stack mentions in order to run the binary, but there's an easier way—`exec`! From our project directory, consider the following:

```
$ hello
zsh: command not found: hello
$ stack exec -- hello
hello world
```

Stack knows which paths any executables might be located in, so using Stack's `exec` command saves you the hassle of typing out a potentially verbose path.

Executable stanzas in Cabal files

Stack created an executable earlier because of the following stanza in the `hello.cabal` file:

```
executable hello
--          [1]
  hs-source-dirs:      src
--    [2]
  main-is:             Main.hs
--    [3]
  default-language:    Haskell2010
--    [4]
  build-depends:       base >= 4.7 && < 5
--    [5]
```

1. This name following the declaration of an *executable* stanza tells Stack or Cabal what to name the binary or executable it creates.
2. Tells this stanza where to look for source code—in this case, the *src* subdirectory.
3. Execution of this binary should begin by looking for a `main` function inside a file named `Main` with the module name `Main`. Note that module names have to match filenames. Your compiler (not just Stack) will reject using a file that isn't a `Main` module as the entry point to executing the program. Also note that it'll look for the `Main.hs` file under all directories you specify in `hs-source-dirs`. Since we specify only one, it'll find this in `src/Main.hs`, which is our only source file right now anyway.
4. Defines the version of the Haskell standard to expect. Not very interesting and doesn't do much—mostly boilerplate but necessary.
5. This is usually a meatier part of any Cabal stanza, whether it's an executable, library, or test suite. This example (`base`) is really the bare minimum or baseline dependency in almost any Haskell project, as you can't really get anything done without the `base` library. We'll show you how to add and install dependencies later.

A sidebar about executables and libraries Our project here only has an executable stanza, which is appropriate for making a command-line application that will be run and used. When we're writing code we want people to be able to reuse in other projects, we need a library stanza in the `.cabal` file, and we need to choose which modules we want to expose. Executables are applications that the operating system will run directly, while software libraries are code arranged in a manner so that they can be reused by the compiler in the building of other libraries and programs.

13.4 Making our project a library

First, we're going to add a library stanza to `hello.cabal`:

```
library
  hs-source-dirs:      src
  exposed-modules:     Hello
  build-depends:       base >= 4.7 && < 5
  default-language:    Haskell2010
```

Then, we're going to create a file located at `src/Hello.hs`:

```
module Hello where

sayHello :: IO ()
sayHello = do
  putStrLn "hello world"
```

Then, we're going to change our `Main` module to use this library function:

```
module Main where

import Hello

main :: IO ()
main = do
  sayHello
```

If we try to build and run this now, it'll work:

```
$ stack build
$ stack exec hello
hello world
```

But what if we had made a separate `exe` directory?

```
$ mkdir exe
$ mv src/Main.hs exe/Main.hs
```

Then, we need to edit the `.cabal` file to let it know our `hello` executable uses the `exe` directory:

```
executable hello
  hs-source-dirs:      exe
  main-is:             Main.hs
  default-language:    Haskell2010
  build-depends:       base >= 4.7 && < 5
```

If you then attempt to build it this time, it will fail:

```
error:
    Could not find module ‘Hello’
    Use -v to see a list of the files
      searched for.
  |
3 | import Hello
  | ^^^^^^^^^^^^
```

We have two paths for fixing this, one better than the other. One way is to simply add `src` to the source directories the executable is permitted to search. But it turns out that Cabal's suggestion here is precisely right. The better way to fix this is to respect the boundaries of the library and executable and instead to add your own library as a dependency:

```
executable hello
  hs-source-dirs:      exe
  main-is:             Main.hs
  default-language:    Haskell2010
  build-depends:       base >= 4.7 && < 5
                     , hello
```

The build will now succeed. This also makes it easier to know when you need to change what is exposed or exported in your library, because you're using your own interface.

13.5 Module exports

By default, when you don't specify any exports in a module, every top-level binding is exported and can be imported by another module. This is the case in our `Hello` module:

```
module Hello where

sayHello :: IO ()
sayHello = do
  putStrLn "hello world"
```

But what happens if we specify an *empty* export list?

```
module Hello
  ()
  where

sayHello :: IO ()
sayHello = do
  putStrLn "hello world"
```

We'll get the following error if we attempt to build it:

```
error: Variable not in scope:
  sayHello :: IO ()
  |
6 | main = sayHello
  |        ^^^^^^^^
```

To fix that explicitly, we add the top-level binding to the export list at the top of the file:

```
module Hello
  ( sayHello )
  where

sayHello :: IO ()
sayHello = do
  putStrLn "hello world"
```

Now the `sayHello` function will be exported. It seems pointless in a module like this, but in bigger projects, it sometimes makes sense to specify your exports in this way.

Exposing modules

First, we'll add a new module with a new `IO` action for our `main` action to run:

```
-- src/DogsRule.hs
module DogsRule
  ( dogs )
  where

dogs :: IO ()
dogs = do
  putStrLn "Who's a good puppy?!"
  putStrLn "YOU ARE!!!!!"
```

Then, we'll change our `Main` module to make use of this function:

```
module Main where

import DogsRule
import Hello

main :: IO ()
main = do
  sayHello
  dogs
```

But if we attempt to build the project, we'll get the following error:

```
error:
    Could not find module ‘DogsRule’
    Use -v to see a list of the files
      searched for.
  |
3 | import DogsRule
  | ^^^^^^^^^^^^^^^
```

As we did earlier with our library stanza, we also need to expose the `DogsRule` module:

```
library
  hs-source-dirs:      src
  exposed-modules:     DogsRule
                     , Hello
  build-depends:       base >= 4.7 && < 5
  default-language:    Haskell2010
```

Now, it should be able to find our very important dog praising.

13.6 More on importing modules

Importing modules brings more functions into scope beyond those available in the standard `Prelude`. Imported modules are top-level declarations. The entities imported as part of those declarations, like other top-level declarations, have scope throughout a module, although they can be shadowed by local bindings. The effect of multiple import declarations is cumulative, but the ordering of import declarations is irrelevant. An entity is in scope for the entire module if it is imported by any of the import declarations.

In previous chapters, we brought functions like `bool` and `toUpper` into scope for exercises by importing the modules they are part of, `Data.Bool` and `Data.Char`, respectively.

Let's refresh our memory of how to do this in GHCi. The `:browse` command allows us to see which functions are included in a named module, while importing the module allows us to use those functions. You can browse modules that you haven't imported yet, which can be

useful if you're not sure which module the function you're looking for is in:

```
Prelude> :browse Data.Bool
bool :: a -> a -> Bool -> a
(&&) :: Bool -> Bool -> Bool
data Bool = False | True
not :: Bool -> Bool
otherwise :: Bool
(||) :: Bool -> Bool -> Bool

Prelude> import Data.Bool


Prelude> :t bool
bool :: a -> a -> Bool -> a
```

In the example above, we use an unqualified import of everything in `Data.Bool`. What if we only want `bool` from `Data.Bool`?

First, we're going to turn off `Prelude`, so that we don't have any of the default imports. We will use another extension when we start GHCi to turn `Prelude` off. You've previously seen how to use language extensions in source files, but now we'll enter `-XNoImplicitPrelude` right when we start our REPL—do this outside of any projects:

```
$ stack ghci --ghci-options -XNoImplicitPrelude
Prelude>
```

We can check that `bool` and `not` are not in scope yet:

```
Prelude> :t bool
<interactive>:1:1: Not in scope: ‘bool’
Prelude> :t not
<interactive>:1:1: Not in scope: ‘not’
```

Next, we'll do a selective import from `Data.Bool`, specifying that we only want to import `bool`:

```
Prelude> import Data.Bool (bool)
Prelude> :t bool
bool :: a -> a -> GHC.Types.Bool -> a
```

```
Prelude> :t not
<interactive>:1:1: Not in scope: ‘not’
```

Now, normally in the `Prelude`, `not` is in scope already, but `bool` is not. So you can see that by turning off `Prelude`, taking its standard functions out of scope, and then importing *only* `bool`, we no longer have the standard `not` function in scope.

You can import one or more functions from a module or library. The syntax is just as we demonstrated with GHCi, but your import declarations have to be at the beginning of a module. Putting `import Data.Char (toUpper)` in the import declarations of a module will ensure that `toUpper`, but not any of the other entities contained in `Data.Char`, is in scope for that module.

For the examples in the next section, you'll want `Prelude` back on, so please restart GHCi before proceeding.

Qualified imports

What if you want to know *where* something you import comes from in the code that uses it? We can use qualified imports to make the names more explicit.

We use the `qualified` keyword in our imports to do this. Sometimes, you'll have stuff with the same name imported from two different modules. Qualifying your imports is a common way of dealing with this. We'll go through an example of how you might use a qualified import:

```
Prelude> import qualified Data.Bool
Prelude> :t bool

• Variable not in scope: bool
• Perhaps you meant ‘Data.Bool.bool’
    (imported from Data.Bool)

Prelude> :t Data.Bool.bool
Data.Bool.bool :: a -> a -> Bool -> a

Prelude> :t Data.Bool.not
Data.Bool.not :: Bool -> Bool
```

In the case of `import qualified Data.Bool`, everything from `Data.Bool` is in scope, but only when accessed with the full `Data.Bool` namespace. We are marking where the functions that we're using come from, which can be useful.

We can also provide aliases or alternate names for our modules when we qualify them, so we don't have to type out the full namespace:

```
Prelude> import qualified Data.Bool as B
Prelude> :t bool

• Variable not in scope: bool
• Perhaps you meant ‘Data.Bool.bool’
    (imported from Data.Bool)

Prelude> :t B.bool
B.bool :: a -> a -> Bool -> a

Prelude> :t B.not
B.not :: Bool -> Bool
```

You can do qualified imports in the import declarations at the beginning of your module, in the same way.

Setting the Prelude prompt When you imported `Data.Bool` as `B` above, you may have seen your prompt change:

```
Prelude> import qualified Data.Bool as B
Prelude B>
```

And if you don't want to unload the imported modules (because you want them all to stay in scope), your prompt could keep growing:

```
Prelude B> import Data.Char
Prelude B Data.Char>
```

Reminder: you can use `:m` to unload the modules, which does, of course, prevent the prompt from growing ever larger, but also, well, unloads the modules, so they're not in scope anymore!

If you want to prevent having an ever-growing prompt, you can use the `:set` command to set the prompt to whatever you prefer:

```
Prelude> :set prompt "Lambda> "
Lambda> import Data.Char
Lambda> :t B.bool
B.bool :: a -> a -> Bool -> a
```

As you can see, `Data.Bool` is still in scope as `B`, but it doesn't show up in our prompt. You can set your `Prelude` prompt permanently, if you wish, by changing it in your GHCi configuration file, but instructions for doing that are outside of the scope of the current chapter.

Intermission: Check your understanding

Here is the import list from one of the modules in a library of Chris's, called `blacktip`:

```
import qualified Control.Concurrent
  as CC
import qualified Control.Concurrent.MVar
  as MV
import qualified Data.ByteString.Char8
  as B
import qualified Data.Locator
  as DL

import qualified Data.Time.Clock.POSIX
  as PSX
import qualified Filesystem
  as FS
import qualified Filesystem.Path.CurrentOS
  as FPC
import qualified Network.Info
  as NI
```

```
import qualified Safe
import Control.Exception (mask, try)
import Control.Monad (forever, when)
import Data.Bits
import Data.Bits.Bitwise (fromListBE)
import Data.List.Split (chunksOf)
import Database.Blacktip.Types
import System.IO.Unsafe (unsafePerformIO)
```

For our purposes right now, it does not matter whether you are familiar with the modules referenced in the import list. Look at the declarations and answer the questions below:

1. What functions are being imported from `Control.Monad`?
2. Which imports are both unqualified and imported in their entirety?
3. From the name, what do you suppose importing the `Types` module brings in?
4. Now, let's compare a small part of `blacktip`'s code to the above import list:

```
writeTimestamp :: MV.MVar ServerState
               -> FPC.FilePath
               -> IO CC.ThreadId
writeTimestamp s path = do
  CC.forkIO go
  where go = forever $ do
          ss <- MV.readMVar s
          mask $ \_ -> do
            FS.writeFile path
            (B.pack (show (ssTime ss)))
          -- sleep for 1 second
          CC.threadDelay 1000000
```

   a) The type signature refers to three aliased imports. What modules are named in those aliases?
   b) Which import does `FS.writeFile` refer to?
   c) Which import does `forever` come from?

13.7 Making our program interactive

Now, we're going to make our program ask for your name, then greet you by name. First, we'll rewrite our `sayHello` function to take an argument:

```
sayHello :: String -> IO ()
sayHello name =
  putStrLn ("Hi " ++ name ++ "!")
```

Note that we parenthesize the `String` argument to `putStrLn`, which is made up of two appending functions using `++`.

Next, we'll change `main` to get the user's name:

```
-- src/Main.hs

main :: IO ()
main = do
  name <- getLine
  sayHello name
  dogs
```

There are a couple of new things here. We're using something called *do* syntax, which is syntactic sugar. We use `do` inside functions that return `IO` in order to sequence side effects using a convenient syntax. Let's decompose what's going on here:

```
 main :: IO ()
 main = do
--     [1]
   name  <-  getLine
-- [4]  [3]    [2]
   sayHello name
--           [5]
   dogs
-- [6]
```

1. The `do` here begins the block.

2. `getLine` has type `IO String`, because it must perform I/O (input/output, side effects) in order to obtain the `String`. `getLine` is what will allow you to enter your name, to be used in the `main` function.

3. `<-` in a `do` block is pronounced *bind*. We'll explain what this is and how it works in the chapters on `Monad` and `IO`.

4. The result of binding (`<-`) over the `IO String` is `String`. We bind it to the variable `name`. Remember, `getLine` has type `IO String`, `name` has type `String`.

5. `sayHello` expects an argument of type `String`, which is the type of `name` but *not* `getLine`.

6. `dogs` expects nothing[4] and is an `IO` action of type `IO ()`, which fits the overall type of `main`.

Let's fire off a build:

```
$ stack build
```

And run the program:

```
$ stack exec hello
```

After you hit enter, the program is going to wait for your input. You'll just see the cursor blinking on the line, waiting for you to enter your name. As soon as you do, and hit enter, it should greet you and then rave about the wonderfulness of a dog.

What if we try to pass getLine to sayHello? If we try to write `main` without the use of `do` syntax, particularly without using `<-`, such as in the following example:

```
main :: IO ()
main = sayHello getLine
```

We'd get the following type error:

[4] Much like actual dogs.

```
$ stack build
[2 of 2] Compiling Main

• Couldn't match expected type
    ‘IO String -> IO ()’
              with actual type ‘IO ()’
• The function ‘sayHello’ is applied to
    one argument,
  but its type ‘IO ()’ has none
  In the expression: sayHello getLine
  In an equation for ‘main’:
    main = sayHello getLine
  |
6 | main = sayHello getLine
  |        ^^
```

This is because `getLine` is an `IO` action with type `IO String`, whereas `sayHello` expects a value of type `String`. We have to use `<-` to bind over the `IO` to get the string that we want to pass to `sayHello`. This will be explained in more detail—a bit more detail later in this chapter and a lot more detail in a later chapter.

Adding a prompt

Let’s make our program a bit easier to use by adding a prompt that tells us our program is expecting input! We need to change `main`:

```
module Main where

import DogsRule
import Hello
import System.IO

main :: IO ()
main = do
  hSetBuffering stdout NoBuffering
  putStr "Please input your name: "
  name <- getLine
  sayHello name
  dogs
```

We do several things here. One is that we use `putStr` instead of `putStrLn` so that our input can be on the same line as our prompt. We also import from `System.IO` so that we can use `hSetBuffering`, `stdout`, and `NoBuffering`. That line of code is so that `putStr` isn't buffered (deferred) and prints immediately. Rebuild and rerun your program, and it should now work like this:

```
$ stack exec hello
Please input your name: julie
Hi julie!
Who's a good puppy?!
YOU ARE!!!!!
```

You can try removing the `NoBuffering` line (that whole first line) from `main` and rebuilding and running your program to see how it changes. We will be using this as part of our hangman game in a bit, but it isn't necessary at this point to understand how the buffering functions work in any detail.

13.8 do syntax and IO

We touched on `do` notation a bit above, but we want to explain a few more things about it. `do` blocks are convenient syntactic sugar that allow for sequencing actions, but because they are only syntactic sugar, they are not, strictly speaking, necessary. They can make blocks of code more readable and also hide some underlying nesting, and that can help you write effectful code before you understand monads and `IO`. So you'll see it a lot in this chapter (and, indeed, you'll see it quite a bit in idiomatic Haskell code).

The `main` executable in a Haskell program must always have the type `IO ()`. The `do` syntax specifically allows us to sequence *monadic actions*. `Monad` is a type class we'll explain in great detail in a later chapter; here, the instance of `Monad` we care about is `IO`. That is why `main` functions are often (not always) `do` blocks.

This syntax also provides a way of naming values returned by monadic IO actions so that they can be used as inputs to actions that happen later in the program. Let's look at a very simple `do` block and try to get a feel for what's happening here:

```
concatUserInput = do
--    [1]
    x1   <- getLine
-- [2]  [3]    [4]
    x2   <- getLine
--          [5]
    return (x1 ++ x2)
--   [6]      [7]
```

1. `do` introduces the block of IO actions.
2. `x1` is a variable representing the value obtained from the IO action `getLine`.
3. `<-` binds the variable on the left to the result of the IO action on the right.
4. `getLine` has the type `IO String` and takes user input as a string value. In this case, the string the user inputs will be the value bound to the `x1` name.
5. `x2` is a variable representing the value obtained from our second `getLine`. As above, it is bound to that value by the `<-`.
6. `return` will be discussed in more detail shortly, but here it is the concluding action of our `do` block.
7. This is the value `return`, well, returns—the conjunction of the two strings we obtain from our two `getLine` actions.

While `<-` is used to bind a variable, it is different from other methods we've seen in earlier chapters for naming and binding variables. This arrow is part of the special `do` sugar and specifically binds a name to the `a` of an `m a` value, where `m` is some monadic structure, in this case `IO`. The `<-` allows us to extract that `a` and name it within the limited scope of the `do` block and use that named value as an input to another expression within that same scope. Each assignment using `<-` creates a new variable rather than mutating an existing variable, because data is immutable.

return

This function really doesn't do a lot, but the purpose it serves is important, given the way monads and `IO` work. It does nothing but return a value, but it returns a value inside a monadic structure:

```
Prelude> :t return
return :: Monad m => a -> m a
```

For our purposes in this chapter, `return` returns a value in `IO`. Because the obligatory type of `main` is `IO ()`, the final value must also have an `IO ()` type, and `return` gives us a way to add no extra function except putting the final value in `IO`. If the final action of a `do` block is `return ()`, that means there is no real value to return at the end of performing the IO actions, but since Haskell programs can't return literally nothing, they return this empty tuple called unit simply to have something to return. That empty tuple will not print to the screen in the REPL, but it's there in the underlying representation.

Let's take a look at `return` in action. Let's say you want to get user input of two characters and test them for equality. You can't do this:

```
twoo :: IO Bool
twoo =  do c  <- getChar
           c' <- getChar
           c == c'
```

Try it, and see what your type error looks like. It should tell you that it can't match the expected type `IO Bool` with the actual type of `c == c'`, which is `Bool`. So, our final line needs to return that `Bool` value in `IO`:

```
twoo :: IO Bool
twoo =  do c  <- getChar
           c' <- getChar
           return (c == c')
```

We put the `Bool` value into `IO` by using `return`. Cool. How about if we have cases where we want to return nothing? We'll reuse the same basic code from above but put an if-then-else expression in our `do` block:

```
main :: IO ()
main =  do c  <- getChar
           c' <- getChar
           if c == c'
            then putStrLn "True"
            else return ()
```

What happens when the two input characters are equal? What happens when they aren't?

Some people have noted that `do` syntax makes it feel like you're doing imperative programming in Haskell. It's important to note that this effectful, imperative style requires having `IO` in our result type. We cannot perform effects without evidence of having done so in the type. `do` is only syntactic sugar, but the monadic syntax we'll cover in a later chapter works in a similar way for monads besides `IO`.

Do notation considered harmful! Just kidding. But sometimes enthusiastic programmers overuse `do` blocks. It is not necessary, and considered bad style, to use `do` in single-line expressions. You will eventually learn to use `>>=` in single-line expressions instead of `do` (there's an example of that in this chapter). Similarly, it is unnecessary to use `do` with functions like `putStrLn` and `print` that already have the effects baked in. In the function above, you can put `do` in front of both `putStrLn` and `return`, and it will work in the same way, but then things start to get messy, and the Haskell ninjas will come and be severely disappointed in you.

13.9 Hangman game

We're ready to build a game. We'll use Stack's `new` command to create this project:

```
$ stack new hangman simple
```

That will generate a directory named `hangman` for you and put some default files into it.

You need a `words` file for getting words from. Most Unix-based operating systems will have a words list located at a directory like the following:

```
$ ls /usr/share/dict/
american-english  british-english
cracklib-small  README.select-wordlist
words  words.pre-dictionaries-common
```

In this case, we'll use the `words` word list, which should be your operating system's default. You may have one in a different place, or you may need to download one. We put it in the working directory at `data/dict.txt`:

```
$ tree .
.
├── LICENSE
├── Setup.hs
├── data
│   └── dict.txt
├── hangman.cabal
├── src
│   └── Main.hs
└── stack.yaml
```

The file is newline-separated and therefore looks like this:

```
$ head data/dict.txt
A
a
aa
aal
aalii
aam
Aani
aardvark
aardwolf
Aaron
```

Edit the `.cabal` file, as follows:

```
name:                hangman
version:             0.1.0.0
synopsis:            Playing Hangman
```

```
homepage:            Chris N Julie
license:             BSD3
license-file:        LICENSE
author:              Chris Allen and Julie Moronuki
maintainer:          haskellbook.com
category:            Game
build-type:          Simple
extra-source-files:  data/dict.txt
cabal-version:       >=1.10
executable hangman
  main-is:             Main.hs
  hs-source-dirs:      src
  build-depends:       base >=4.7 && <5
                     , random
                     , split
  default-language:    Haskell2010
```

The important bit here is that we use two libraries: `random` and `split`. Normally, you'd do version ranges for your dependencies, like you see with `base`, but we leave the versions of `random` and `split` unassigned, because they do not change much. The primary and only source file is in `src/Main.hs`.

13.10 Step One: Importing modules

```
-- src/Main.hs

module Main where

import Control.Monad (forever) -- [1]
import Data.Char (toLower) -- [2]
import Data.Maybe (isJust) -- [3]
import Data.List (intersperse) -- [4]
import System.Exit (exitSuccess) -- [5]
import System.IO (BufferMode(NoBuffering),
                  hSetBuffering,
                  stdout) -- [6]
import System.Random (randomRIO) -- [7]
```

Here, the imports are enumerated in the source code. For your version of this project, you don't need to add the enumerating comments. All modules listed below are part of the main `base` library that comes with your GHC install, unless otherwise noted.

1. We're using `forever` from `Control.Monad` to make an infinite loop. A couple points to note:
   a) You don't *have* to use `forever` to do this, but we're going to.
   b) You are not expected to understand what it does or how it works, exactly. Basically, it allows us to execute a function over and over again, infinitely, or until we cause the program to exit or fail, instead of evaluating once and then stopping.
2. We will use `toLower` from `Data.Char` to convert all the characters of our string to lowercase:

   ```
   Prelude> import Data.Char (toLower)
   Prelude> toLower 'A'
   'a'
   ```

   Be aware that if you pass a character that doesn't have a sensible lowercase, `toLower` will kick the same character back out:

   ```
   Prelude> toLower ':'
   ':'
   ```

3. We will use `isJust` from `Data.Maybe` to determine whether every character in our puzzle has been discovered already or not:

   ```
   Prelude> import Data.Maybe (isJust)
   Prelude> isJust Nothing
   False
   Prelude> isJust (Just 10)
   True
   ```

   We will combine this with `all`, a standard function in the `Prelude`. `all` is a function that answers the question, "given a function that will return `True` or `False` for each element, does it return `True` for *all* of them?"

```
Prelude> all even [2, 4, 6]
True
Prelude> all even [2, 4, 7]
False
Prelude> xs = [Just 'd', Nothing, Just 'g']
Prelude> all isJust xs
False
Prelude> ys = [Just 'd', Just 'o', Just 'g']
Prelude> all isJust ys
True
```

The function `all` has the type:

```
Foldable t => (a -> Bool) -> t a -> Bool
```

We haven't explained the `Foldable` type class. For your purposes, you can assume it's a set of operations for types that can be folded in a manner conceptually similar to the list type but which don't *necessarily* contain more than one value (or any values at all) the way a list or similar datatype does. We can make the type more specific by asserting a type signature:

```
Prelude> :t all :: (a -> Bool) -> [a] -> Bool
all :: (a -> Bool) -> [a] -> Bool
```

This will work for any type that has a `Foldable` instance, such as `Maybe`. Try entering the following examples into a file and then type checking them in your REPL:

```
maybeAll :: (a -> Bool)
         -> Maybe a
         -> Bool
maybeAll = all
```

Note the type variables used, and experiment independently:

```
eitherAll :: (a -> Bool)
          -> Either b a
          -> Bool
eitherAll = all
```

But it will not work if the datatype doesn't have an instance of `Foldable`:

```
badAll :: (a -> Bool)
       -> (b -> a)
       -> Bool
badAll = all
```

When you type check the above, you should get the following error:

```
• No instance for (Foldable ((->) b1))
    arising from a use of ‘all’
• In the expression: all
  In an equation for ‘badAll’: badAll = all
```

4. We use `intersperse` from `Data.List` to... intersperse elements in a list. In this case, we're putting spaces between the characters guessed so far by the player. You may remember we used `intersperse` back in Chapter 8, on recursion, to put hyphens in our "Numbers Into Words" exercise:

```
Prelude> import Data.List (intersperse)
Prelude> intersperse ' ' "Blah"
"B l a h"
```

Conveniently, the type of `intersperse` says nothing about characters or strings, so we can use it with lists containing elements of any type:

```
Prelude> :t intersperse
intersperse :: a -> [a] -> [a]

Prelude> intersperse 0 [1, 1, 1]
[1,0,1,0,1]
```

5. We use `exitSuccess` from `System.Exit` to exit successfully—no errors, we're simply done. We indicate whether it was a success or not so our operating system knows whether an error occurred.

Note that if you evaluate `exitSuccess` in the REPL, it'll report that an exception occurred. In a normally running program that doesn't catch the exception, it'll end your whole program.

6. We use `System.IO`'s handle for standard output (or `stdout`) and for setting the buffering mode in the terminal. This is to avoid potential problems some environments can have with interactive terminal programs, usually due to rendering output immediately.

7. We use `randomRIO` from `System.Random` to select a word from our dictionary at random. `System.Random` is in the library `random`. Once again, you'll need to have the library in scope for your REPL to be able to load it. Once it's in scope, we can use `randomRIO` to get a random number. You can see from the type signature that it takes a tuple as an argument, but it uses the tuple as a range from which to select a random item:

```
Prelude> import System.Random
Prelude System.Random> :t randomRIO
randomRIO :: Random a => (a, a) -> IO a
Prelude System.Random> randomRIO (0, 5)
4
Prelude System.Random> randomRIO (1, 100)
71
Prelude System.Random> randomRIO (1, 100)
12
```

Later, we will use this random number generation to produce a random index into a word list, to provide a means of selecting random words for our puzzle.

13.11 Step Two: Generating a word list

For clarity's sake, we're using a type synonym to declare what we mean by `[String]` in our types. Later, we'll show you a version that's even more explicit using `newtype`. We also use `do` syntax to read the contents of our dictionary into a variable named `dict`. We use the `lines` function to split the big blob string we read from the file into

a list of string values, each representing a single line. Each line is a single word, so our result is the `WordList`:

```
type WordList = [String]

allWords :: IO WordList
allWords = do
  dict <- readFile "data/dict.txt"
  return (lines dict)
```

Let's take a moment to look at `lines`, which splits strings at the newline marks and returns a list of strings:

```
Prelude> lines "aardvark\naaron"
["aardvark","aaron"]
Prelude> length $ lines "aardvark\naaron"
2
Prelude> let s = "aardvark\naaron\nwoot"
Prelude> length $ lines s
3
Prelude> lines "aardvark aaron"
["aardvark aaron"]
Prelude> length $ lines "aardvark aaron"
1
```

Note that this does something similar but different from `words`, which splits by spaces (ostensibly between words) *and* newlines:

```
Prelude> words "aardvark aaron"
["aardvark","aaron"]
Prelude> words "aardvark\naaron"
["aardvark","aaron"]
```

The next part of building our word list for our puzzle is to set upper and lower bounds for the size of words we'll use in the puzzles. Feel free to change them, if you want:

```
minWordLength :: Int
minWordLength = 5
```

```
maxWordLength :: Int
maxWordLength = 9
```

The next thing we're going to do is take the output of `allWords` and filter it to fit the length criteria we define above. That will give us a shorter list of words to use in the puzzles:

```
gameWords :: IO WordList
gameWords = do
  aw <- allWords
  return (filter gameLength aw)
  where gameLength w =
          let l = length (w :: String)
          in     l >= minWordLength
              && l <  maxWordLength
```

We next need to write a pair of functions that will pull a random word out of our word list for us, so that the puzzle player doesn't know what the word will be. We're going to use the `randomRIO` function we mention above to facilitate that. We'll pass `randomRIO` a tuple of zero (the first indexed position in our word list) and the number that is the length of our word list minus one. Why minus one?

We have to subtract one from the length of the word list in order to index it, because `length` starts counting from 1, but an index of the list starts from 0. A list of length 5 does not have a member indexed at position 5—it has inhabitants at positions 0–4 instead:

```
Prelude> [1..5] !! 4
5
Prelude> [1..5] !! 5
*** Exception: Prelude.(!!): index too large
```

In order to get the last value in the list, then, we must ask for the member in the position of the length of the list minus one:

```
Prelude> myList = [1..5]
Prelude> length myList
5
Prelude> myList !! length myList
*** Exception: Prelude.!!: index too large
```

```
Prelude> myList !! (length myList - 1)
5
```

The next two functions work together to pull a random word out of the `gameWords` list we created above. Roughly speaking, `randomWord` generates a random index number based on the length of a word list, `wl`, and then selects the member of that list that is at that indexed position and returns an `IO String`. Given what you know about `randomRIO` and indexing, you should be able to supply the tuple argument to `randomRIO` yourself:

```
randomWord :: WordList -> IO String
randomWord wl = do
  randomIndex <- randomRIO ( , )
--        fill this part in ^^^
  return $ wl !! randomIndex
```

The second function, `randomWord'` binds the `gameWords` list to the `randomWord` function, so that the random word we're getting is from that list. We're going to delay a full discussion of the `>>=` operator, known as "bind," until we get to the chapter on `Monad`. For now, we can say that, as we said about `do` syntax, the *bind* function allows us to sequentially compose actions, such that a value generated by the first becomes an argument to the second:

```
randomWord' :: IO String
randomWord' = gameWords >>= randomWord
```

Now that we have a word list, we turn our attention to the building of an interactive game using it.

13.12 Step Three: Making a puzzle

Our next step is to formulate the core game play. We need a way to hide the word from the player (while giving them an indication of how many letters it has) and create a means of asking for letter guesses, determining if the guessed letter is in the word, putting it in the word if it is and putting it into an "already guessed" list if it's not, and determining when the game ends.

We start with a datatype for our puzzle. The puzzle is a *product* of a `String`, a list of `Maybe Char`, and a list of `Char`:

```
data Puzzle =
  Puzzle String [Maybe Char] [Char]
--        [1]       [2]       [3]
```

1. The word we're trying to guess.
2. The characters we've filled in so far.
3. The letters we've guessed so far.

Next, we're going to write an instance of the type class `Show` for our datatype `Puzzle`. You may recall that `show` allows us to print human-readable, stringy things to the screen, which is obviously something we have to do to interact with our game. But we want it to print our puzzle a certain way, so we define this instance.

Notice how the argument to `show` lines up with our datatype definition above. Now, `discovered` refers to our list of `Maybe Char`, and `guessed` is what we name our list of `Char`, but we've done nothing with the `String` itself:

```
instance Show Puzzle where
  show (Puzzle _ discovered guessed) =
    (intersperse ' ' $
     fmap renderPuzzleChar discovered)
    ++ " Guessed so far: " ++ guessed
```

This is going to show us two things as part of our puzzle: the list of `Maybe Char`, which is the string of characters we have correctly guessed, and the rest of the characters of the puzzle word represented by underscores, interspersed with spaces; and a list of `Char` that reminds us which characters we've already guessed. We'll talk about `renderPuzzleChar` below.

First, we're going to write a function that will take our puzzle word and turn it into a list of `Nothing`. This is the first step in hiding the word from the player. We're going to ask you to write this one yourself, using the following information:

- We've given you a type signature. Your first argument is a `String`, which will be the word that is in play. It will return a value of type `Puzzle`. Remember that the `Puzzle` type is a product of three things.
- Your first value in the output will be the same string as the argument to the function.
- The second value will be the result of mapping a function over that `String` argument. Consider using `const` in the mapped function, as it will always return its first argument, no matter what its second argument is.
- For purposes of this function, the final argument of `Puzzle` is an empty list.

Go for it:

```
freshPuzzle :: String -> Puzzle
freshPuzzle = undefined
```

Now we need a function that looks at the `Puzzle String` and determines whether the character you guessed is an element of that string. Here are some hints:

- This is going to need two arguments, and one of those is of type `Puzzle`, which is a product of three types. But for the purpose of this function, we only care about the first argument to `Puzzle`.
- We can use underscores to signal that there are values we don't care about and tell the function to ignore them. Whether you use underscores to represent the arguments you don't care about or go ahead and use names for them won't affect the result of the function. It does, however, keep your code a bit cleaner and easier to read by explicitly signaling which arguments you care about in a given function.
- The standard function `elem` works like this:

```
Prelude> :t elem
elem :: Eq a => a -> [a] -> Bool
Prelude> elem 'a' "julie"
```

```
False
Prelude> elem 3 [1..5]
True
```

So, here you go:

```
charInWord :: Puzzle -> Char -> Bool
charInWord = undefined
```

The next function is similar to the one you just wrote, but this time we don't care whether the `Char` is part of the `String` argument—this time, we want to check and see if it is an element of the `guessed` list.

You've totally got this:

```
alreadyGuessed :: Puzzle -> Char -> Bool
alreadyGuessed = undefined
```

OK, so far we have ways to choose a word that we're trying to guess and determine if a guessed character is part of that word or not. But we need a way to hide the rest of the word from the player while they're guessing. Computers are a bit dumb, after all, and can't figure out how to keep secrets on their own. Back when we defined our `Show` instance for this puzzle, we `fmap`ped a function called `renderPuzzleChar` over our second `Puzzle` argument. Let's work on that function next.

The goal here is to use `Maybe` to permit two different outcomes. It will be mapped over a string in the type class instance, so this function works on only one character at a time. If that character has not been correctly guessed yet, it's a `Nothing` value and should appear on the screen as an underscore. If the character has been guessed, we want to display that character so the player can see which positions they've correctly filled:

```
Prelude> renderPuzzleChar Nothing
'_'
Prelude> renderPuzzleChar (Just 'c')
'c'
Prelude> n = Nothing
Prelude> xs = [n, Just 'h', n, Just 'e', n]
Prelude> fmap renderPuzzleChar xs
"_h_e_"
```

Your turn. Remember, you don't need to do the mapping part of it here:

```
renderPuzzleChar :: Maybe Char -> Char
renderPuzzleChar = undefined
```

The next bit is a touch tricky. The point is to insert a correctly guessed character into the string. Although none of the components here are new to you, they're put together in a somewhat dense manner, so we're going to unpack it (obviously, when you type this into your own file, you do not need to add the enumerations):

```
fillInCharacter :: Puzzle -> Char -> Puzzle
fillInCharacter (Puzzle word
--                      [1]
                filledInSoFar s) c =
--                               [2]
  Puzzle word newFilledInSoFar (c : s)
--             [     3     ]

  where zipper guessed wordChar guessChar =
--       [4]     [5]     [6]       [7]
          if wordChar == guessed
          then Just wordChar
          else guessChar
--        [      8     ]
        newFilledInSoFar =
--              [9]
          zipWith (zipper c)
            word filledInSoFar
--        [        10        ]
```

1. The first argument is our `Puzzle` with its three arguments, with `s` representing the list of characters already guessed.
2. The `c` is our `Char` argument and is the character the player guessed on this turn.
3. Our result is the `Puzzle` with `filledInSoFar` replaced by `newFilledInSoFar`. To accumulate the characters guessed by the player, we cons `c` onto the list of characters each time a guess is made.

4. `zipper` is a combining function for deciding how to handle the character in the word, what's been guessed already, and the character that was just guessed. If the current character in the word is equal to what the player guesses, then we go ahead and return `Just wordChar` to fill in that spot in the puzzle. Otherwise, we kick the `guessChar` back out. We kick `guessChar` back out, because it might either be a previously correctly guessed character *or* a `Nothing` that has not been guessed correctly this time nor in the past.
5. `guessed` is the character the player guesses.
6. `wordChar` is for the characters in the puzzle word—not the ones they've guessed or not guessed, but the characters in the original word that they're trying to guess.
7. `guessChar` is the character that the player has guessed.
8. This if-then-else expression checks to see if the guessed character is one of the word characters. If it is, it wraps it in a `Just`, because our puzzle word is a list of `Maybe` values.
9. `newFilledInSoFar` is the new state of the puzzle, which uses `zipWith` and the `zipper` combining function to fill in characters in the puzzle. The `zipper` function is first applied to the character the player just guessed, because that doesn't change. Then, it's zipped across two lists. One list is `word`, which is the word the user is trying to guess. The second list, `filledInSoFar`, is the puzzle state we're starting with of type `[Maybe Char]`. That's telling us which characters in `word` have been guessed.
10. Now, we're going to make our `newFilledInSoFar` by using `zipWith`. You may remember this from Chapter 9, on lists. It's going to zip the `word` with the `filledInSoFar` values, while applying the `zipper` function from just above it to those values as it does so.

Next, we have this big `do` block with a case expression and each case also has a `do` block inside it. Why not, right?

First, we tell the player what they guessed. The case expression is to give different responses based on whether the guessed character:

- Had already been guessed previously.

- Is in the word and needs to be filled in.
- Was not previously guessed but also isn't in the puzzle word.

Despite the initial appearance of complexity, most of this is syntax you've seen before, and you can look through it step-by-step and see what's going on:

```
handleGuess :: Puzzle -> Char -> IO Puzzle
handleGuess puzzle guess = do

  putStrLn $ "Your guess was: " ++ [guess]
  case (charInWord puzzle guess
       , alreadyGuessed puzzle guess) of

   (_, True) -> do
     putStrLn "You already guessed that\
              \ character, pick \
              \ something else!"
     return puzzle

   (True, _) -> do
     putStrLn "This character was in the\
              \ word, filling in the word\
              \ accordingly"
     return (fillInCharacter puzzle guess)

   (False, _) -> do
     putStrLn "This character wasn't in\
               \ the word, try again."
     return (fillInCharacter puzzle guess)
```

All right, next we need to devise a way to stop the game after a certain number of guesses. Hangman games normally stop only after a certain number of *incorrect* guesses, but for the sake of simplicity here, we're stopping after a set number of guesses, whether they're correct or not. Again, the syntax here should be comprehensible to you from what we've done so far:

```
gameOver :: Puzzle -> IO ()
gameOver (Puzzle wordToGuess _ guessed) =
  if (length guessed) > 7 then
    do putStrLn "You lose!"
       putStrLn $
         "The word was: " ++ wordToGuess
       exitSuccess
  else return ()
```

Notice that the way it's written says you lose and exits the game once you've guessed seven characters, even if the final (seventh) guess is the final letter that completes the word. There are, of course, ways to modify that to make it more the way you'd expect a hangman game to go, and we encourage you to play with this code.

Next, we need to provide a way to exit after winning the game. We showed you how the combination of `isJust` and `all` works earlier in the chapter, and you can see that in action here. Recall that our puzzle word is a list of `Maybe` values, so when each character is represented by a `Just Char` rather than a `Nothing`, you win the game and we exit:

```
gameWin :: Puzzle -> IO ()
gameWin (Puzzle _ filledInSoFar _) =
  if all isJust filledInSoFar then
    do putStrLn "You win!"
       exitSuccess
  else return ()
```

Next is the action for running a game. Here, we use `forever` to execute this series of actions indefinitely:

```
runGame :: Puzzle -> IO ()
runGame puzzle = forever $ do
  gameOver puzzle
  gameWin puzzle
  putStrLn $
    "Current puzzle is: " ++ show puzzle
  putStr "Guess a letter: "
```

```
  guess <- getLine
  case guess of
   [c] -> handleGuess puzzle c >>= runGame
   _   ->
     putStrLn "Your guess must\
             \ be a single character"
```

And, finally, `main` brings everything together: it gets a word from the word list we generated, generates a fresh puzzle, and then executes the `runGame` actions we saw above, until such time as you guess all the characters in the word correctly or have made seven guesses, whichever comes first:

```
main :: IO ()
main = do
  hSetBuffering stdout NoBuffering
  word <- randomWord'
  let puzzle =
        freshPuzzle (fmap toLower word)
  runGame puzzle
```

13.13 Adding a newtype

Another way you could modify your code above and gain, perhaps, more clarity in places is with the use of newtype:

```
-- replace this type synonym
-- type WordList = [String]

newtype WordList =
  WordList [String]
  deriving (Eq, Show)

allWords :: IO WordList
allWords = do
  dict <- readFile "data/dict.txt"
  return $ WordList (lines dict)
```

```
gameWords :: IO WordList
gameWords = do
  (WordList aw) <- allWords
  return $ WordList (filter gameLength aw)
  where gameLength w =
          let l = length (w :: String)
          in     l > minWordLength
              && l < maxWordLength

randomWord :: WordList -> IO String
randomWord (WordList wl) = do
  randomIndex <-
    randomRIO (0, (length wl) - 1)
  return $ wl !! randomIndex
```

13.14 Chapter exercises

Hangman game logic

You may have noticed when you were playing with the hangman game, that there are some weird things about its game logic:

- Although it can play with words up to nine characters long, you only get to guess seven characters.
- It ends the game after seven guesses, whether they were correct or incorrect.
- If your seventh guess supplies the last letter in the word, it may still tell you that you lost.
- It picks some very strange words that you didn't suspect were even in the dictionary.

These make it unlike hangman as you might have played it in the past. Ordinarily, only incorrect guesses count against you, so you can make as many correct guesses as you need to fill in the word. Modifying the game so that it either gives you more guesses before the game ends or only uses shorter words (or both) involves only a couple of uncomplicated steps.

A bit more complicated but worth attempting as an exercise is changing the game so that, as with normal hangman, only incorrect guesses count toward the guess limit.

Modifying code

1. Ciphers: Open your ciphers module, and modify it so that the Caesar and Vigenère ciphers work with user input.

2. Here is a very simple, short block of code. Notice it has a `forever` that will make it keep running, over and over again. Load it into your REPL, and test it out. Then, refer back to the chapter, and modify it to exit successfully after a `False` result:

```
import Control.Monad

palindrome :: IO ()
palindrome = forever $ do
  line1 <- getLine
  case (line1 == reverse line1) of
    True -> putStrLn "It's a palindrome!"
    False -> putStrLn "Nope!"
```

3. If you try using `palindrome` on a sentence such as "Madam I'm Adam," you may notice that it doesn't work. Modifying the above so that it works on sentences, too, involves several steps. You may need to refer back to previous examples in the chapter to get ideas for proper ordering and nesting. You may wish to import `Data.Char` to use the function `toLower`. Have fun.

4.
```
type Name = String
type Age = Integer

data Person = Person Name Age deriving Show

data PersonInvalid =
    NameEmpty
  | AgeTooLow
  | PersonInvalidUnknown String
  deriving (Eq, Show)

mkPerson :: Name
         -> Age
         -> Either PersonInvalid Person
mkPerson name age
  | name /= "" && age > 0 =
      Right $ Person name age
  | name == "" = Left NameEmpty
  | not (age > 0) = Left AgeTooLow
  | otherwise =
      Left $ PersonInvalidUnknown $
        "Name was: " ++ show name ++
        " Age was: " ++ show age
```

Your job is to write the following function *without* modifying the code above:

```
gimmePerson :: IO ()
gimmePerson = undefined
```

Since `IO ()` is about the least informative type imaginable, we'll tell you what it should do:

a) It should prompt the user for a name and age input.
b) It should attempt to construct a `Person` value using the name and age the user enters. You'll need the `read` function for the age, because it's an `Integer` rather than a `String`.
c) If it constructs a successful person, it should print `"Yay! Successfully got a person: "` followed by the `Person` value.

d) If it gets an error value, it should report that an error occurred and print the error.

13.15 Follow-up resources

1. The Haskell Tool Stack.
   `https://github.com/commercialhaskell/stack`
2. Chris Allen. *How I Start: Haskell*.
   `https://bitemyapp.com/blog/how-i-start-haskell/`
3. Cabal FAQ.
   `https://www.haskell.org/cabal/FAQ.html`
4. Cabal User Guide.
   `https://www.haskell.org/cabal/users-guide/`
5. Paul Hudak, John Peterson, and Joseph Fasel. *A Gentle Introduction to Haskell*. Section 11. "Modules."
   `https://www.haskell.org/tutorial/patterns.html`

Chapter 14

Testing

> We've tended to forget that no computer will ever ask a new question.
>
> Grace Murray Hopper

14.1 Testing

This chapter, like the one before it, is more focused on practical matters rather than writing Haskell code, per se. We will be covering two testing libraries (there are others) and how and when to use them. You will not be writing much of the code in the chapter on your own; instead, please follow along by entering it into files as directed (you will learn more if you type rather than copy and paste). At the end of the chapter, there are a number of exercises that ask you to write your own tests, for practice.

Testing is a core part of the working programmer's toolkit, and Haskell is no exception. Well-specified types can enable programmers to avoid many obvious and tedious tests that might otherwise be necessary to maintain in untyped programming languages, but there's still a lot of value to be obtained in executable specifications. This chapter will introduce you to testing methods for Haskell.

This chapter will cover:

- The whats and whys of testing.
- Using the testing libraries `hspec` and `QuickCheck`.
- A bit of fun with Morse code.

14.2 A quick tour of testing for the uninitiated

When we write Haskell, we rely on the compiler to judge for us whether our code is well-formed. That prevents a great number of errors, but it does not prevent them all. It is still possible to write well-typed code that doesn't perform as expected, and runtime errors can still occur. That's where testing comes in.

In general, tests allow you to state an expectation and then verify that the result of an operation meets that expectation. They allow you to verify that your code will do what you want when executed.

For the sake of simplicity, we'll say there are two broad categories of testing: unit testing and property testing. Unit testing tests the smallest atomic units of software independently of one another. Unit testing allows the programmer to check that each function is performing the task it is meant to do. You assert that when the code runs with a specified input, the result is equal to the result you want.

Spec testing is a somewhat newer version of unit testing. Like unit testing, it tests specific functions independently and asks you to assert that, when given the declared input, the result of the operation will be equal to the desired result. When you run the test, the computer checks that the expected result is equal to the actual result and everyone moves on with their day. Some people prefer spec testing to unit testing, because spec testing is more often written in terms of assertions that are in human-readable language. This can be especially valuable if non-programmers need to be able to read and interpret the results of the tests—they can read the English-language results of the tests and, in some cases, write tests themselves.

Haskell provides libraries for both unit and spec testing. We'll focus on specification testing with the `hspec` library in this chapter, but `HUnit` is also available. One limitation to unit and spec testing is that they test atomic units of code independently, so they do not verify that all the pieces work properly *together*.

Property testing is a different beast. This kind of testing was pioneered in Haskell, because the type system and straightforward logic of the language lend themselves to property tests, but it has since been adopted by other languages, as well. Property tests test the formal properties of programs without requiring formal proofs, by allowing you to express a truth-valued, universally quantified (that is, will apply to all cases) function—usually equality—that is checked against randomly generated inputs.

The inputs are generated randomly by the standard functions inside the `QuickCheck` library we use for property testing. This relies on the type system to know what kinds of data to generate. The default setting is for 100 inputs to be generated, giving you 100 results. If it fails any one of these, then you know your program doesn't have the specified property. If it passes, you can't be positive it will never fail, because the data are randomly generated—there could be a weird edge case out there that will cause your software to fail anyway. `QuickCheck` is cleverly written to be as thorough as possible and will usually check the most common edge cases (for example, empty lists and the `maxBound` and `minBound` values of the types in question, where appropriate). You can also change the setting so that it runs more tests.

Property testing is fantastic for ensuring that you've met the min-

imum requirements to satisfy laws, such as the laws of monads or basic associativity. It is not appropriate for all programs, though, as it is not useful for times when there are no assertable, truth-valued properties of the software.

14.3 Conventional testing

We are going to use the library `hspec`[1] to demonstrate a test case, but we're not going to explain `hspec` *deeply*. The current chapter will equip you with a means of writing tests for your code later, but it's not necessary to understand the details of how the library works to do that. Some of the concepts `hspec` leans on, such as functor, applicative, and monad, are independently covered later.

First, let's come up with a test case for addition. Generally, we want to make a Cabal package, even for small experiments. Having a permanent project for experiments can eliminate some of this overhead, but we'll assume you haven't done this yet and start a small project now:

```
-- addition.cabal
name:                addition
version:             0.1.0.0
license-file:        LICENSE
author:              Chicken Little
maintainer:          sky@isfalling.org
category:            Text
build-type:          Simple
cabal-version:       >=1.10

library
  exposed-modules:     Addition
  ghc-options:         -Wall -fwarn-tabs
  build-depends:       base >=4.7 && <5
                     , hspec
  hs-source-dirs:      .
  default-language:    Haskell2010
```

[1] `http://hackage.haskell.org/package/hspec`

Note that we've specified the `hspec` dependency but not a version range for it. You'll probably want whatever the newest version of it is, but you can probably get away with not specifying it for now.

Next, we'll make the `Addition` module (in `exposed-modules`, above) in the same directory as our Cabal file. This is why the `hs-source-dirs` option in the library stanza is set to `.` above—this is the convention for referring to the current directory.

For now, we'll write a simple placeholder function to make sure everything's working:

```
-- Addition.hs
module Addition where

sayHello :: IO ()
sayHello = putStrLn "hello!"
```

Then, you can create an empty `LICENSE` file, so the build doesn't complain:

```
$ touch LICENSE
```

Your local project directory should now look like this, before you have run any Stack commands:

```
$ tree
.
├── Addition.hs
└── addition.cabal
└── LICENSE
```

The next steps are to initialize the Stack file for describing what snapshot of Stackage we'll use:

```
$ stack init
```

Then, we'll want to build our project, which will also install the dependencies we need:

```
$ stack build
```

If that succeeds, let's fire up a REEEEEEEPL and see if we can call `sayHello`:

```
$ stack ghci

...some noise about configuring,
  loading packages, etc...

Ok, one module loaded.
Prelude> sayHello
hello!
```

If you got here, you've got a working test bed for making a simple test case in `hspec`!

Truth according to Hspec

Next, we'll add the import of `hspec`'s primary module:

```
module Addition where

import Test.Hspec

sayHello :: IO ()
sayHello = putStrLn "hello!"
```

Note that *all* of your imports must occur after the module has been declared and before any expressions have been defined in your file. Otherwise, you may encounter an error. Here are a couple of examples:

```
module Addition where

sayHello :: IO ()
sayHello = putStrLn "hello!"

import Test.Hspec
```

Here, we put an import after at least one declaration. The compiler parser doesn't have a means of recognizing this specific mistake, so it can't tell you properly what the error is:

```
Prelude> :r
[1 of 1] Compiling Addition

Addition.hs:6:1: error: parse error on
  input ‘import’
  |
6 | import Test.Hspec
  | ^^^^^^
Failed, no modules loaded.
```

What else could go wrong? Well, we might have the package `hspec` installed but not included in our `build-depends` for our project. Note that you'll need to quit and reopen the REPL if you make any changes to your `.cabal` file to reproduce this error or fix a mistake:

```
$ stack build
...noise...

    Could not find module ‘Test.Hspec’
    Use -v to see a list of the files
      searched for.
  |
3 | import Test.Hspec
  | ^^^^^^^^^^^^^^^^^

...other noise...

Process exited with code: ExitFailure 1
```

If you change anything in order to test these error modes, you'll need to add `hspec` back to your `build-depends` and reinstall it. If `hspec` is listed in your dependencies, `stack build` will set you right.

Assuming everything is in order and `Test.Hspec` is being imported, we can do a little exploration. We can use the `:browse` command to get a listing of types from a module and get a thousand-foot-view of what it offers:

```
Prelude> :browse Test.Hspec
context :: String
```

```
        -> SpecWith a
        -> SpecWith a
example :: Expectation -> Expectation
specify :: Example a
        => String
        -> a
        -> SpecWith (Arg a)
...list goes on for awhile...
```

`:browse` is more useful when you already have some familiarity with the library and how it works. When you're using an unfamiliar library, documentation is easier to digest. Good documentation explains how important pieces of a library work and gives examples of their use. This is especially valuable when encountering new concepts. As it happens, `hspec` has some pretty good documentation on its website.[2]

Our first Hspec test

Let's add a test assertion to our module now. If you glance at the documentation, you'll see that our example isn't very interesting, but we'll make it somewhat more interesting soon:

```
module Addition where

import Test.Hspec

main :: IO ()
main = hspec $ do
  describe "Addition" $ do
    it "1 + 1 is greater than 1" $ do
      (1 + 1) > 1 `shouldBe` True
```

We've asserted in both English and code that 1 + 1 should be greater than 1, and that is what `hspec` will test for us. You may recognize the `do` notation from the previous chapter. As we said then, this syntax allows us to sequence monadic actions. In the previous chapter, the monad in question was `IO`.

[2] `http://hspec.github.io/`

Here, we're nesting multiple `do` blocks. The types of the `do` blocks passed to `hspec`, `describe`, and `it` aren't `IO ()` but something more specific to `hspec`. They result in `IO ()` in the end, but there are other monads involved. We haven't covered monads yet, and this works fine without understanding precisely how it works, so let's just roll with it for now.

Note that you'll get warnings about the `Num a => a` literals getting defaulted to `Integer`. You can ignore this or add explicit type signatures—it's up to you. With the above code in place, we can load or reload our module and run `main` to see the test results:

```
Prelude> main


Addition
  1 + 1 is greater than 1

Finished in 0.0041 seconds
1 example, 0 failures
```

OK, so what's happening here? Basically, `hspec` runs your code and verifies that the arguments you pass to `shouldBe` are equal. Let's look at the types:

```
shouldBe :: (Eq a, Show a)
         => a -> a -> Expectation
```

Contrast this with:

```
(==) :: Eq a => a -> a -> Bool
```

In a sense, it's an augmented `==` embedded in `hspec`'s model of the universe. It needs the `Show` instance in order to render a value. That is, the `Show` instance allows `hspec` to show you the result of the tests, not just return a `Bool` value.

Let's add another test, one that reads a little differently:

```
main :: IO ()
main = hspec $ do
  describe "Addition" $ do
    it "1 + 1 is greater than 1" $ do
      (1 + 1) > 1 `shouldBe` True
    it "2 + 2 is equal to 4" $ do
      2 + 2 `shouldBe` 4
```

Modify your `describe` block so that it looks like the one above, and run it in the REPL:

```
Prelude> main

Addition
  1 + 1 is greater than 1
  2 + 2 is equal to 4

Finished in 0.0004 seconds
2 examples, 0 failures
```

For fun, we'll look back to something you wrote early in the book and write a short `hspec` test for it. Back in Chapter 8, on recursion, we wrote our own division function that looked like this:

```
dividedBy :: Integral a => a -> a -> (a, a)
dividedBy num denom = go num denom 0
  where go n   d count
        | n < d = (count, n)
        | otherwise =
            go (n - d) d (count + 1)
```

We want to test it to see that it works as it should. To keep things simple, we add `dividedBy` to our `Addition.hs` file and then rewrite the `hspec` tests that were already there. We want to test that the function is both subtracting the correct number of times and keeping an accurate count of that subtraction and also that it's telling us the correct remainder, so we'll give `hspec` two things to test for:

```
main :: IO ()
main = hspec $ do
  describe "Addition" $ do
    it "15 divided by 3 is 5" $ do
      dividedBy 15 3 `shouldBe` (5, 0)
    it "22 divided by 5 is\
       \ 4 remainder 2" $ do
      dividedBy 22 5 `shouldBe` (4, 2)
```

That's it. When we reload `Addition.hs` in our REPL, we can test our division function:

```
*Addition> main

Addition
  15 divided by 3 is 5
  22 divided by 5 is 4 remainder 2

Finished in 0.0012 seconds
2 examples, 0 failures
```

Hurrah! We can do arithmetic!

Intermission: Short exercise

In the Chapter Exercises at the end of Chapter 8, you were given this exercise:

Write a function that multiplies two numbers using recursive summation. The type should be `(Eq a, Num a) => a -> a -> a`, although, depending on how you do it, you might also consider adding an `Ord` constraint.

If you still have your answer, great! If not, rewrite it, and then write `hspec` tests for it.

The above examples demonstrate the basics of writing individual tests to test particular values. If you'd like to see a more developed example, you could refer to Chris's library, `Bloodhound`.[3]

[3] `https://github.com/bitemyapp/bloodhound`

14.4 Enter QuickCheck

`hspec` does a nice job with spec testing, but we're Haskell users—we're never satisfied! `hspec` can only prove something about particular values. Can we get assurances that are stronger, something closer to proofs? As it happens, we can.

`QuickCheck` was the first library to offer what is today called property testing. `hspec` testing is more like what is known as unit testing—the testing of individual units of code—whereas property testing is done with the assertion of laws or properties.

First, we'll need to add `QuickCheck` to our `build-depends`. Open your `.cabal` file and add it. Be sure to capitalize `QuickCheck` (unlike `hspec`, which begins with a lowercase h). It should already be installed, as `hspec` has `QuickCheck` as a dependency, but you may need to reinstall it (`stack build`). Then, open a new `stack ghci` session.

`hspec` has `QuickCheck` integration out of the box, so once that is done, add the following to your module:

```
-- with your imports
import Test.QuickCheck


-- to the same describe block as the others
    it "x + 1 is always\
       \ greater than x" $ do
      property $ \x -> x + 1 > (x :: Int)
```

If we don't assert the type of `x` in the property test, the compiler won't know which concrete type to use, and we'll see a message like this:

```
• Ambiguous type variable ‘a0’ arising from
    a use of ‘property’
  prevents the constraint ‘(Arbitrary a0)’
    from being solved.
...
• Ambiguous type variable ‘a0’ arising from
    a use of ‘+’
  prevents the constraint ‘(Num a0)’
    from being solved.
```

```
...
• Ambiguous type variable ‘a0’ arising from
    a use of ‘>’
  prevents the constraint ‘(Ord a0)’
    from being solved.
```

Avoid this by asserting a concrete type, for example, `(x :: Int)`, in the property.

Assuming all is well, when we run it, we'll see something like the following:

```
Prelude> main

Addition
  1 + 1 is greater than 1
  2 + 2 is equal to 4
  x + 1 is always greater than x

Finished in 0.0067 seconds
3 examples, 0 failures
```

What's being hidden a bit by `hspec` is that `QuickCheck` tests *many* values to see if your assertions hold for all of them. It does this by randomly generating values of the type you tell it to expect. So, it'll keep feeding our function random `Int` values to see if the property is ever false. The number of tests `QuickCheck` runs defaults to 100.

Arbitrary instances

`QuickCheck` relies on a type class called `Arbitrary` and a `newtype` called `Gen` for generating its random data.

`arbitrary` is a value of type `Gen`:

```
Prelude> :t arbitrary
arbitrary :: Arbitrary a => Gen a
```

This is a way to set a default generator for a type. When you use the `arbitrary` value, you have to specify the type to dispatch the right type class instance, as types and type class instances form unique

pairings. But this is just a value. How do we see a list of values of the correct type?

We can use `sample` and `sample'` from the `Test.QuickCheck` module in order to see some random data.

This prints each value on a new line:

```
Prelude> :t sample
sample :: Show a => Gen a -> IO ()
```

This one returns a list:

```
Prelude> :t sample'
sample' :: Gen a -> IO [a]
```

The `IO` is necessary, because it's using a global resource of random values to generate the data. A common way to generate pseudorandom data is to have a function that, given some input "seed" value, returns a value and another seed value for generating a different value. You can bind the two actions together, as we explained in the last chapter, to pass a new seed value each time and keep generating seemingly random data. In this case, however, we're not doing that. Here, we're using `IO` so that the function that generates our data can return a different result each time (not something pure functions are allowed to do) by pulling from a global resource of random values. If this doesn't make much sense at this point, it will be clearer once we cover monads and even more so once we cover `IO`.

We use the `Arbitrary` type class in order to provide a generator for `sample`. It isn't a terribly principled type class, but it is popular and useful for this. We say it is unprincipled, because it has no laws and nothing specific it's supposed to do. It's a convenient way of plucking a canonical generator for `Gen a` out of thin air without having to know where it comes from. If it feels a bit like **MAGICK** at this point, that's fine. It is, a bit, and the inner workings of `Arbitrary` are not worth fussing over right now.

As you'll see later, this isn't necessary if you have a `Gen` value ready to go already. `Gen` is a newtype with a single type argument. It exists for wrapping up a function to generate pseudorandom values. The function takes an argument that is usually provided by some kind of random value generator to give you a pseudorandom value of that

type, assuming it's a type that has an instance of the `Arbitrary` type class.

And this is what we get when we use the `sample` functions. We use the `arbitrary` value but specify the type, so that it gives us a list of random values of that type:

```
Prelude> sample (arbitrary :: Gen Int)
0
-2
-1
4
-3
4
2
4
-3
2
-4
Prelude> sample (arbitrary :: Gen Double)
0.0
0.13712502861905426
2.9801894108743605
-8.960645064542609
4.494161946149201
7.903662448338119
-5.221729489254451
31.64874305324701
77.43118278366954
-539.7148886375935
26.87468214215407
```

If you run `sample arbitrary` directly in GHCi without specifying a type, it will default the type to `()` and give you a very nice list of empty tuples. If you try loading an unspecified `sample arbitrary` from a source file, though, you will get an affectionate message from GHC about having an ambiguous type. Try it if you like. GHCi has somewhat different rules for default types than GHC does.

We can specify our own data for generating `Gen` values. In this example, we'll specify a trivial function that always returns a 1 of type `Int`:

```
-- trivial generator of values
trivialInt :: Gen Int
trivialInt = return 1
```

You may remember `return` from the previous chapter, as well. Here, it provides an expedient way to construct a function. In the last chapter, we noted that it doesn't do a whole lot except return a value inside of a monad. Before, we were using it to put a value into `IO`, but it's not limited to use with that monad:

```
return :: Monad m => a -> m a

-- when m is Gen:
return :: a -> Gen a
```

Putting 1 into the `Gen` monad constructs a generator that always returns the same value, 1.

So, what happens when we sample data from this?

```
Prelude> sample' trivialInt
[1,1,1,1,1,1,1,1,1,1,1]
```

Notice now that our value isn't `arbitrary` for some type, but the `trivialInt` value we define above. That generator always returns 1, so all `sample'` can return for us is a list of 1s.

Let's explore different means of generating values:

```
oneThroughThree :: Gen Int
oneThroughThree = elements [1, 2, 3]
```

Try loading that via your `Addition` module and asking for a `sample` set of random `oneThroughThree` values:

```
*Addition> sample' oneThroughThree
[2,3,3,2,2,1,2,1,1,3,3]
```

Yep, it gives us random values from only that limited set. At this time, each number in that set has the same chance of showing up in our random data set. We could tinker with those odds by having a list with repeated elements to give those elements a higher probability of showing up in each generation:

```
oneThroughThree :: Gen Int
oneThroughThree =
  elements [1, 2, 2, 2, 2, 3]
```

Try running `sample'` again with this set, and see if you notice the difference. You may not, of course, because due to the nature of probability, there is at least some chance that 2 won't show up any more than it did with the previous sample.

Next, we'll use `choose` and `elements` from the `QuickCheck` library as generators of values:

```
genBool :: Gen Bool
genBool = choose (False, True)

genBool' :: Gen Bool
genBool' = elements [False, True]

genOrdering :: Gen Ordering
genOrdering = elements [LT, EQ, GT]

genChar :: Gen Char
genChar = elements ['a'..'z']
```

You should enter all these into your `Addition` module, load them into your REPL, and play with getting lists of sample data for each.

Our next examples are a bit more complex:

```
genTuple :: (Arbitrary a, Arbitrary b)
         => Gen (a, b)
genTuple = do
  a <- arbitrary
  b <- arbitrary
  return (a, b)
```

```
genThreeple :: (Arbitrary a, Arbitrary b,
                Arbitrary c)
            => Gen (a, b, c)
genThreeple = do
  a <- arbitrary
  b <- arbitrary
  c <- arbitrary
  return (a, b, c)
```

Here's how to use generators when they have polymorphic type arguments. Remember that if you leave the types unspecified, the extended defaulting behavior of GHCi will (helpfully?) pick `()` for you. Outside of GHCi, you'll get an error about an ambiguous type—we covered some of this when we explained type classes earlier:

```
Prelude> sample genTuple
((),())
((),())
((),())
```

Here, it's defaulting the `a` and `b` to `()`. We can get more interesting output if we tell it what we expect `a` and `b` to be. Note, it'll always pick 0 and 0.0 for the first numeric values:

```
Prelude> type G = Gen (Int, Float)
Prelude> sample (genTuple :: G)
(0,0.0)
(-1,0.2516606)
(3,0.7800742)
(5,-61.62875)
```

We can ask for lists and characters or anything with an instance of the `Arbitrary` type class:

```
Prelude> type G = Gen ([()], Char)
Prelude> sample (genTuple :: G)
([],'\STX')
([()],'X')
([],'?')
([],'\137')
```

```
([(),()],'\DC1')
([(),()],'z')
```

You can use `:info Arbitrary` in your REPL to see which instances are available.

We can also generate arbitrary `Maybe` and `Either` values:

```
genEither :: (Arbitrary a, Arbitrary b)
          => Gen (Either a b)
genEither = do
  a <- arbitrary
  b <- arbitrary
  elements [Left a, Right b]

-- equal probability
genMaybe :: Arbitrary a => Gen (Maybe a)
genMaybe = do
  a <- arbitrary
  elements [Nothing, Just a]

-- What QuickCheck does so
-- you get more Just values
genMaybe' :: Arbitrary a => Gen (Maybe a)
genMaybe' = do
  a <- arbitrary
  frequency [ (1, return Nothing)
            , (3, return (Just a))]

-- frequency :: [(Int, Gen a)] -> Gen a
```

For now, you should play with this in the REPL. It will become useful to know about later on.

Using QuickCheck without Hspec

We can also use `QuickCheck` without `hspec`. In that case, we no longer need to specify `x` in our expression, because the type of `prop_additionGreater` provides for it. Thus, we can rewrite our previous example, as follows:

```
prop_additionGreater :: Int -> Bool
prop_additionGreater x = x + 1 > x

runQc :: IO ()
runQc = quickCheck prop_additionGreater
```

For now, we don't need to worry about how `runQc` does its work. It's a generic function, like `main`, that signals that it's time to do stuff. Specifically, in this case, it's time to perform the `QuickCheck` tests.

Now, when we run it in the REPL, instead of the `main` we were calling with `hspec`, we'll call `runQc`, which will call on `QuickCheck` to test the property we define. When we run `QuickCheck` directly, it reports how many tests it runs:

```
Prelude> runQc
+++ OK, passed 100 tests.
```

What happens if we assert something untrue?

```
prop_additionGreater x = x + 0 > x
```

```
Prelude> :r
[1 of 1] Compiling Addition
Ok, one module loaded.
Prelude> runQc
*** Failed! Falsifiable (after 1 test):
0
```

Conveniently, `QuickCheck` doesn't only tell us that our test fails, but it tells us the first input it encounters that it fails on. If you try to keep running it, you may notice that the value that it fails on is always 0. A while ago, we said that `QuickCheck` has some built-in cleverness and tries to ensure that common error boundaries will always get tested. The input 0 is a frequent point of failure, so `QuickCheck` tries to ensure that it is always tested (when appropriate, given the types, etc.).

14.5 Morse code

In the interest of playing with testing, we'll work through an example project in which we translate text to and from Morse code. We're

going to start a new project for this. When you use `stack new project-name` to start a new project instead of `stack init` for an existing project, it automatically generates a file called `Setup.hs` that looks like this:

```
import Distribution.Simple
main = defaultMain
```

This isn't terribly important. You rarely need to modify or do anything at all with the `Setup.hs` file, and usually you shouldn't touch it at all. Occasionally, you may need to edit it for certain tasks, so it is good to recognize that it's there.

Next, as always, let's get our `.cabal` file configured properly. Some of this will be generated automatically when you run `stack new project-name`, but you'll have to add to what is generated for you, being careful about things like capitalization and indentation:

```
name:                morse
version:             0.1.0.0
license-file:        LICENSE
author:              Chris Allen
maintainer:          cma@bitemyapp.com
category:            Text
build-type:          Simple
cabal-version:       >=1.10

library
  exposed-modules:     Morse
  ghc-options:         -Wall -fwarn-tabs
  build-depends:       base >=4.7 && <5
                     , containers
                     , QuickCheck
  hs-source-dirs:      src
  default-language:    Haskell2010

executable morse
  main-is:             Main.hs
  ghc-options:         -Wall -fwarn-tabs
  hs-source-dirs:      src
  build-depends:       base >=4.7 && <5
```

```
                    , containers
                    , morse
                    , QuickCheck
  default-language:    Haskell2010

test-suite tests
  ghc-options: -Wall -fno-warn-orphans
  type: exitcode-stdio-1.0
  main-is: tests.hs
  hs-source-dirs: tests
  build-depends:       base
                    , containers
                    , morse
                    , QuickCheck
  default-language:    Haskell2010
```

Don't forget to capitalize the `QuickCheck` dependency properly! With that set up and ready for us, the next step is to make our `src` directory and the file called `Morse.hs` as our "exposed module:"

```
-- src/Morse.hs

module Morse
       ( Morse
       , charToMorse
       , morseToChar
       , stringToMorse
       , letterToMorse
       , morseToLetter
       ) where

import qualified Data.Map as M

type Morse = String
```

Whoa, there—what's all that stuff after the module name? That is a list of everything this module will export. We talked a bit about this in the previous chapter but didn't make use of it. In the hangman

game, we had all our functions in one file, so nothing needed to be exported.

Nota bene You don't have to specify exports in this manner. By default, the entire module is exposed and can be imported by any other module. If you want to export everything in a module, then specifying exports is unnecessary. However, it can help, when managing large projects, to specify what will get used by other modules (and, by exclusion, what will not) as a way of documenting your intent. In this case, we have exported here more than what we will import into `Main`. We will only need two functions for `Main`. You'll see this in a bit. We could go back and remove the things we didn't specifically import from the above export list, but we haven't now, to give you an idea of the process we're going through putting our project together.

Turning words into code

We are also using a qualified import of `Data.Map`. We covered this type of import somewhat in the previous chapter. We qualify the import and name it `M` so that we can use that `M` as a prefix for the functions we're using from that package. That will help us keep track of where the functions come from and also avoid same-name clashes with `Prelude` functions, but without requiring us to tediously type `Data.Map` as a prefix to each function name.

We'll talk more about `Map` as a data structure later in the book. For now, we can understand it as being a balanced binary tree, where each node is a pairing of a key and a value. The key is an index for the value—a marker of how to find the value in the tree. The key must be orderable (that is, it must have an `Ord` instance), much like our binary tree functions earlier, such as `insert`, need an `Ord` instance. Maps can be more efficient than lists, because you do not have to search linearly through a bunch of data. Because the keys are ordered and the tree is balanced, searching through the binary tree divides the search space in half each time you go "left" or "right." You compare the key to the index of the current node to determine if you need to go left (less), right (greater), or if you've arrived at the node for your value (equals).

You can see below why we use a `Map` instead of a simple list. We want to make a list of pairs, where each pair includes both the English-

language character and its Morse code representation. We define our transliteration table thus:

```
letterToMorse :: (M.Map Char Morse)
letterToMorse = M.fromList [
      ('a', ".-")
    , ('b', "-...")
    , ('c', "-.-.")
    , ('d', "-..")
    , ('e', ".")

    , ('f', "..-.")
    , ('g', "--.")
    , ('h', "....")
    , ('i', "..")
    , ('j', ".---")
    , ('k', "-.-")

    , ('l', ".-..")
    , ('m', "--")
    , ('n', "-.")
    , ('o', "---")
    , ('p', ".--.")
    , ('q', "--.-")

    , ('r', ".-.")
    , ('s', "...")
    , ('t', "-")
    , ('u', "..-")
    , ('v', "...-")
    , ('w', ".--")

    , ('x', "-..-")
    , ('y', "-.--")
    , ('z', "--..")
    , ('1', ".----")
    , ('2', "..---")
```

```
    , ('3', "...--")
    , ('4', "....-")
    , ('5', ".....")

    , ('6', "-....")
    , ('7', "--...")
    , ('8', "---..")
    , ('9', "----.")
    , ('0', "-----")
    ]
```

Note that we use `M.fromList`—the `M` prefix tells us this comes from `Data.Map`. We're using a `Map` to associate characters with their Morse code representations. `letterToMorse` is the definition of the `Map` we'll use to look up the codes for individual characters.

Next, we write a few functions that allow us to convert a Morse character to an English character and vice versa and also functions to do the same for strings:

```
morseToLetter :: M.Map Morse Char
morseToLetter =
  M.foldrWithKey (flip M.insert) M.empty
                 letterToMorse

charToMorse :: Char -> Maybe Morse
charToMorse c =
  M.lookup c letterToMorse

stringToMorse :: String -> Maybe [Morse]
stringToMorse s =
  sequence $ fmap charToMorse s

morseToChar :: Morse -> Maybe Char
morseToChar m =
  M.lookup m morseToLetter
```

Notice we use `Maybe` in three of those functions, since not every `Char` that could potentially occur in a `String` has a Morse representation.

The Main event

Next, we want to set up a `Main` module that will handle our Morse code conversions. Note that it's going to import a bunch of things, some of which we covered in the last chapter and some we have not. Since we will not be going into the specifics of how this code works, we won't discuss those imports here. It is, however, important to note that one of our imports is our `Morse.hs` module from above:

```
-- src/Main.hs

module Main where

import Control.Monad (forever, when)
import Data.List (intercalate)
import Data.Traversable (traverse)
import Morse (stringToMorse, morseToChar)
import System.Environment (getArgs)
import System.Exit (exitFailure,
                    exitSuccess)
import System.IO (hGetLine, hIsEOF, stdin)
```

As we said, we're not going to explain this part in detail. We encourage you to do your best reading and interpreting of it, but it's quite dense, and this chapter isn't about this code—it's about the tests. We're cargo-culting a bit here, which we don't like to do, but we're doing it so that we can focus on the testing. Type this all into your `Main` module—first the function to convert a text string of alphabetical characters to a string of Morse dots and dashes:

```
convertToMorse :: IO ()
convertToMorse = forever $ do
  weAreDone <- hIsEOF stdin
  when weAreDone exitSuccess

  -- otherwise, proceed
  line <- hGetLine stdin
  convertLine line
```

```
  where
    convertLine line = do
      let morse = stringToMorse line
      case morse of
       (Just str)
         -> putStrLn
            (intercalate " " str)
       Nothing
         -> do
           putStrLn $ "ERROR: " ++ line
           exitFailure
```

Now, add the function to convert from Morse:

```
convertFromMorse :: IO ()
convertFromMorse = forever $ do
  weAreDone <- hIsEOF stdin
  when weAreDone exitSuccess

  -- otherwise, proceed
  line <- hGetLine stdin
  convertLine line

  where
    convertLine line = do
      let decoded :: Maybe String
          decoded =
            traverse morseToChar
                     (words line)
      case decoded of
       (Just s) -> putStrLn s
       Nothing  -> do
         putStrLn $ "ERROR: " ++ line
         exitFailure
```

And now our obligatory `main`:

```
main :: IO ()
main = do
  mode <- getArgs
  case mode of
   [arg] ->
     case arg of
      "from" -> convertFromMorse
      "to"   -> convertToMorse
      _      -> argError
   _ -> argError

  where argError = do
          putStrLn "Please specify the\
                   \ first argument\
                   \ as being 'from' or\
                   \ 'to' morse,\
                   \ such as: morse to"
          exitFailure
```

Make sure it's all working

One way we can make sure everything is working for us from the command line is by using `echo`. If this is familiar to you, and you feel comfortable with it, go ahead and try this:

```
$ stack build
$ echo "hi" | stack exec morse to
.... ..

$ echo ".... .." | stack exec morse from
hi
```

If you'd like to find out where Stack puts the executable, you can use `stack exec which morse` on Mac and Linux. You can also use `stack install` to ask Stack to build (if needed) and copy the binaries from your project into a common directory. On Mac and Linux that will be `.local/bin` in your home directory. This location was chosen partly to respect XDG[4] guidelines.

[4] `https://wiki.archlinux.org/index.php/Xdg_user_directories`

Otherwise, load this module into your GHCi REPL and give it a try to ensure everything compiles and seems to be in working order. It'll be helpful to fix any type or syntax errors now, before we start trying to run the tests.

Time to test!

Now, we need to write our test suite. We have those in their own directory and file. We will again call the module `Main`, but note the filename (the name per se isn't important, but it must agree with the test file you name in your Cabal configuration for this project):

```
-- tests/tests.hs

module Main where

import qualified Data.Map as M
import Morse
import Test.QuickCheck
```

We have many fewer imports for this, which should all already be familiar to you.

Now, we set up our generators for ensuring that the random values `QuickCheck` uses to test our program are sensible for our Morse code program:

```
allowedChars :: [Char]
allowedChars = M.keys letterToMorse

allowedMorse :: [Morse]
allowedMorse = M.elems letterToMorse

charGen :: Gen Char
charGen = elements allowedChars

morseGen :: Gen Morse
morseGen = elements allowedMorse
```

We saw `elements` briefly, above. It takes a list of some type—in these cases, our lists of allowed characters and Morse characters—and

chooses a `Gen` value from the values in that list. Because `Char` includes thousands of characters that have no legitimate equivalent in Morse code, we need to write our own custom generators.

Next, we write up the property we want to check. We want to check that when we convert something to Morse code and then back again, it comes out as the same string we started out with:

```
prop_thereAndBackAgain :: Property
prop_thereAndBackAgain =
  forAll charGen
  (\c -> ((charToMorse c)
    >>= morseToChar) == Just c)

main :: IO ()
main = quickCheck prop_thereAndBackAgain
```

This is how your setup should look when you have done all of this:

```
$ tree
.
├── LICENSE
├── Setup.hs
├── morse.cabal
├── src
│   ├── Main.hs
│   └── Morse.hs
├── stack.yaml
└── tests
    └── tests.hs
```

Testing the Morse code

Now that our conversions seem to be working, let's run our tests to make sure. The property we're testing is that we get the same string after we convert it to Morse and back again. Let's load up our tests by opening a REPL from our main project directory:

```
$ stack ghci morse:tests
```

```
...noise noise noise...

Ok, one module loaded.
Prelude>
```

Sweet. Stack loads everything for us and even builds our dependencies if needs be. Let's see what happens:

```
Prelude> main
+++ OK, passed 100 tests.
```

The test generates 100 random Morse code conversions and makes sure they are always equal once you have converted to and then from Morse code. This gives you a pretty strong assurance that your program is correct and will perform as expected for any input value.

14.6 Arbitrary instances

One of the more important parts of becoming an expert user of `QuickCheck` is learning to write instances of the `Arbitrary` type class for your datatypes. It's a somewhat unfortunate but still necessary convenience for your code to integrate cleanly with `QuickCheck` code. It's initially a bit confusing for beginners, because it compacts a few different concepts and solutions to problems into a single type class.

Babby's first Arbitrary

First, we'll begin with a maximally simple `Arbitrary` instance for the `Trivial` datatype:

```
module Main where

import Test.QuickCheck

data Trivial =
  Trivial
  deriving (Eq, Show)
```

```
trivialGen :: Gen Trivial
trivialGen =
  return Trivial

instance Arbitrary Trivial where
  arbitrary = trivialGen
```

The `return` is necessary to return `Trivial` in the `Gen` monad:

```
main :: IO ()
main = do
  sample trivialGen
```

Let's take a sample:

```
Prelude> sample trivialGen
Trivial
Trivial
Trivial
Trivial
Trivial
Trivial
Trivial
Trivial
Trivial
Trivial
Trivial
```

Although it's impossible to see the point with `Trivial` by itself, `Gen` values are generators of random values from which `QuickCheck` gets its test values.

Identity crisis

This one is a little different. It will produce random values even if the `Identity` structure itself doesn't and cannot vary:

```
data Identity a =
  Identity a
  deriving (Eq, Show)

identityGen :: Arbitrary a =>
               Gen (Identity a)
identityGen = do
  a <- arbitrary
  return (Identity a)
```

We're using the `Gen` monad to pluck a single value of type `a` out of the air, embed it in `Identity`, then return it as part of the `Gen` monad. We know this is weird, but if you do it ten or twenty times you might start to like it.

We'll reuse the original `identityGen` we wrote. We can make it the default generator for the `Identity` type by making it the `arbitrary` value in the `Arbitrary` instance:

```
instance Arbitrary a =>
         Arbitrary (Identity a) where
  arbitrary = identityGen

identityGenInt :: Gen (Identity Int)
identityGenInt = identityGen
```

We're making a generator suitable for sampling by making the type argument of `Identity` unambiguous for testing with the `sample` function. Your output in the terminal should look something like this:

```
Prelude> sample identityGenInt
Identity 0
Identity (-1)
Identity 2
Identity 4
Identity (-3)
Identity 5
Identity 3
Identity (-1)
```

```
Identity 12
Identity 16
Identity 0
```

You should also be able to change the concrete type of the type argument to `Identity` and generate different types of sample values.

Arbitrary products

`Arbitrary` instances for product types get a teensy bit more interesting, but they're really an extension of what we did for `Identity`:

```
data Pair a b =
  Pair a b
  deriving (Eq, Show)

pairGen :: (Arbitrary a,
            Arbitrary b) =>
           Gen (Pair a b)
pairGen = do
  a <- arbitrary
  b <- arbitrary
  return (Pair a b)
```

We will reuse our `pairGen` function as the `arbitrary` value in the instance:

```
instance (Arbitrary a,
          Arbitrary b) =>
         Arbitrary (Pair a b) where
  arbitrary = pairGen

pairGenIntString :: Gen (Pair Int String)
pairGenIntString = pairGen
```

And now we can generate some sample values:

```
Pair 0 ""
Pair (-2) ""
Pair (-3) "26"
```

```
Pair (-5) "B\NUL\143:\254\SO"
Pair (-6) "\184*\239\DC4"
Pair 5 "\238\213=J\NAK!"
Pair 6 "Pv$y"
Pair (-10) "G|J^"
Pair 16 "R"
Pair (-7) "("
Pair 19 "i\ETX]\182\ENQ"
```

Ah, the beauty of random `String` values.

Greater than the sum of its parts

Writing `Arbitrary` instances for sum types is a bit more interesting, still. First, make sure the following is included in your imports:

```
import Test.QuickCheck.Gen (oneof)
```

Sum types represent disjunction, so with a sum type like `Sum`, we need to represent the exclusive possibilities in our `Gen`. One way to do that is to pull out as many `arbitrary` values as you require for the cases of your sum type. We have two data constructors in this sum type, so we'll want two `arbitrary` values. Then we'll repack them into `Gen` values, resulting in a value of type `[Gen a]` that can be passed to `oneof`:

```
data Sum a b =
    First a
  | Second b
  deriving (Eq, Show)

-- equal odds for each
sumGenEqual :: (Arbitrary a,
                Arbitrary b) =>
               Gen (Sum a b)
sumGenEqual = do
  a <- arbitrary
  b <- arbitrary
  oneof [return $ First a,
         return $ Second b]
```

The `oneof` function will create a `Gen a` from a list of `Gen a` by giving each value an equal probability. From there, you're delegating to the `Arbitrary` instances of the types `a` and `b`:

```
sumGenCharInt :: Gen (Sum Char Int)
sumGenCharInt = sumGenEqual
```

We specify which `Arbitrary` instances to use for `a` and `b` and do a test run:

```
Prelude> sample sumGenCharInt
First 'P'
First '\227'
First '\238'
First '.'
Second (-3)
First '\132'
Second (-12)
Second (-12)
First '\186'
Second (-11)
First '\v'
```

Where sum types get even more interesting is that you can choose a different weighting of probabilities than an equal distribution. Consider this snippet of the `Maybe Arbitrary` instance from the `QuickCheck` library:

```
instance Arbitrary a =>
         Arbitrary (Maybe a) where
  arbitrary =
    frequency [(1, return Nothing),
               (3, liftM Just arbitrary)]
```

It's making an arbitrary `Just` value three times more likely than a `Nothing` value, because the former is more likely to be interesting and useful, but you still want to try shaking things out with a `Nothing` from time to time.

Accordingly, we can assign a 10 times higher probability to our `First` data constructor in a different `Gen` for `Sum`:

```
sumGenFirstPls :: (Arbitrary a,
                   Arbitrary b) =>
                  Gen (Sum a b)
sumGenFirstPls = do
  a <- arbitrary
  b <- arbitrary
  frequency [(10, return $ First a),
             (1, return $ Second b)]

sumGenCharIntFirst :: Gen (Sum Char Int)
sumGenCharIntFirst = sumGenFirstPls
```

With that modified version, you'll find `Second` values are much less common:

```
First '\208'
First '\242'
First '\159'
First 'v'
First '\159'
First '\232'
First '3'
First 'l'
Second (-16)
First 'x'
First 'Y'
```

One of the key insights here is that the `Arbitrary` instance for a datatype doesn't have to be the only way to generate or provide random values of your datatype for `QuickCheck` tests. You can offer alternative `Gen` values for your type with interesting or useful behaviors, as well.

14.7 Chapter exercises

Now it's time to write some tests of your own. You could write tests for most of the exercises you've done in the book, but whether you'd want to use `hspec` or `QuickCheck` depends on what you're trying to test. We've tried to simplify things a bit by telling you which to use for

these exercises, but, as always, we encourage you to experiment on your own.

Validating numbers into words

Remember the "numbers into words" exercise in Chapter 8? You'll be writing tests to validate the functions you wrote:

```
module WordNumberTest where

import Test.Hspec
import WordNumber
  (digitToWord, digits, wordNumber)

main :: IO ()
main = hspec $ do
  describe "digitToWord" $ do
    it "returns zero for 0" $ do
      digitToWord 0 `shouldBe` "zero"
    it "returns one for 1" $ do
      print "???"

  describe "digits" $ do
    it "returns [1] for 1" $ do
      digits 1 `shouldBe` [1]
    it "returns [1, 0, 0] for 100" $ do
      print "???"

  describe "wordNumber" $ do
    it "one-zero-zero given 100" $ do
      wordNumber 100
        `shouldBe` "one-zero-zero"
    it "nine-zero-zero-one for 9001" $ do
      print "???"
```

Fill in the test cases that print question marks. If you think of additional tests you could perform, add them.

Using QuickCheck

Test some basic properties using `QuickCheck`:

1. ```
   -- for a function
   half x = x / 2


   -- this property should hold
   halfIdentity = (*2) . half
   ```

2. ```
   import Data.List (sort)

   -- for any list you apply sort to,
   -- this property should hold
   listOrdered :: (Ord a) => [a] -> Bool
   listOrdered xs =
     snd $ foldr go (Nothing, True) xs
     where go _ status@(_, False) = status
           go y (Nothing, t) = (Just y, t)
           go y (Just x, t) = (Just y, x >= y)
   ```

3. Now, we'll test the associative and commutative properties of addition:

   ```
   plusAssociative x y z =
     x + (y + z) == (x + y) + z
   plusCommutative x y =
     x + y == y + x
   ```

   Keep in mind, these properties won't hold for types based on IEEE-754 floating point numbers, such as `Float` or `Double`.

4. Now do the same for multiplication.

5. We mentioned in one of the first chapters that there are some laws involving the relationships of `quot` to `rem` and `div` to `mod`. Write `QuickCheck` tests to prove them:

   ```
   -- quot rem
   (quot x y) * y + (rem x y) == x


   (div x y) * y + (mod x y) == x
   ```

6. Is the `^` operation associative? Is it commutative? Use `QuickCheck` to see if the computer can contradict such an assertion.

7. Test that reversing a list twice is the same as the identity of the original list:

   ```
   reverse . reverse == id
   ```

8. Write a property for the definition of `$`:

   ```
   f $ a = f a


   f . g = \x -> f (g x)
   ```

9. See if these two functions are equal:

   ```
   foldr (:) == (++)


   foldr (++) [] == concat
   ```

10. Hmm. Is that so?

    ```
    f n xs = length (take n xs) == n
    ```

11. Finally, this is a fun one. You may remember we had you compose `read` and `show` one time to complete a "round trip." Well, now you can test that it works:

    ```
    f x = (read (show x)) == x
    ```

Failure

Find out why this property fails:

```
-- for a function
square x = x * x

-- Why does this property not hold?
-- Examine the type of sqrt.
squareIdentity = square . sqrt
```

Hint: Read about floating point arithmetic and precision if you're unfamiliar with it.

Idempotence

Idempotence refers to a property of some functions for which the result value does not change beyond the initial application. If you apply the function once, it returns a result, and applying the same function to that value won't ever change it. You might think of a list that you sort: once you sort it, the sorted list will remain the same after applying the same sorting function to it again. It's already sorted, so new applications of the sort function won't change it further.

Use `QuickCheck` and the following helper functions to demonstrate idempotence for the following:

```
twice f = f . f
fourTimes = twice . twice
```

1. ```
   f x =
     (capitalizeWord x
     == twice capitalizeWord x)
     &&
     (capitalizeWord x
     == fourTimes capitalizeWord x)
   ```

2. ```
   f' x =
     (sort x
     == twice sort x)
     &&
     (sort x
     == fourTimes sort x)
   ```

Make a Gen random generator for the datatype

We demonstrate in this chapter how to make `Gen` generators for different datatypes. We are so certain you enjoyed that, we are going to ask you to do it for some new datatypes:

1. Equal probabilities for each:

   ```
   data Fool =
       Fulse
     | Frue
     deriving (Eq, Show)
   ```

2. 2/3s chance of `Fulse`, 1/3 chance of `Frue`:

```
data Fool =
    Fulse
  | Frue
  deriving (Eq, Show)
```

Hangman testing

Next, you should go back to the hangman project from the previous chapter and write tests. The kinds of tests you can write at this point will be limited due to the interactive nature of the game. However, you can test the functions. Focus your attention on testing the following:

```
fillInCharacter :: Puzzle -> Char -> Puzzle
fillInCharacter (Puzzle word
                 filledInSoFar s) c =
  Puzzle word newFilledInSoFar (c : s)

  where zipper guessed wordChar guessChar =
          if wordChar == guessed
          then Just wordChar
          else guessChar

        newFilledInSoFar =
          let zd = (zipper c)
          in zipWith zd word filledInSoFar
```

And:

```
handleGuess :: Puzzle -> Char -> IO Puzzle
handleGuess puzzle guess = do
  putStrLn $ "Your guess was: " ++ [guess]
```

```
case (charInWord puzzle guess
    , alreadyGuessed puzzle guess) of
 (_, True) -> do
   putStrLn "You already guessed that\
            \ character, pick\
            \ something else!"
   return puzzle

 (True, _) -> do
   putStrLn "This character was in the\
            \ word, filling in the\
            \ word accordingly"
   return (fillInCharacter puzzle guess)

 (False, _) -> do
   putStrLn "This character wasn't in\
            \ the word, try again."
   return (fillInCharacter puzzle guess)
```

Refresh your memory on what those are supposed to do, and then test to make sure they do it.

Validating ciphers

As a final exercise, create `QuickCheck` properties that verify that your Caesar and Vigenère ciphers return the same data after encoding and decoding a string.

14.8 Definitions

1. *Unit testing* is a method in which you test the smallest parts of an application possible. These units are individually and independently scrutinized for desired behaviors. Unit testing is better automated, but it can also be done manually via a human entering inputs and verifying outputs.

2. *Property testing* is a testing method where a subset of a large input space is validated, usually against a property or law some code should abide by. In Haskell, this is usually done with

`QuickCheck`, which facilitates the random generation of inputs and the definition of properties to be verified. Common properties that are checked using property testing are things like identity, associativity, isomorphism, and idempotence.

3. When we say an operation or function is `idempotent` or satisfies *idempotence*, we mean that applying it multiple times doesn't produce a different result from the first time. One example is multiplying by one or zero. You always get the same result as the first time you multiply any number by one or zero.

14.9 Follow-up resources

1. Pedro Vasconcelos. *An introduction to QuickCheck testing*. `https://web.archive.org/web/20150925050956/https://www.fpcomplete.com/user/pbv/an-introduction-to-quickcheck-testing`
2. Koen Claessen and John Hughes. *QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs*.
3. Pedro Vasconcelos. *Verifying a Simple Compiler Using Property-based Random Testing*. `https://web.archive.org/web/20180403102125/http://www.dcc.fc.up.pt:80/dcc/Pubs/TReports/TR13/dcc-2013-06.pdf`

Chapter 15

Monoid, Semigroup

> Simplicity does not precede complexity, but follows it.
>
> Alan Perlis

15.1 Monoids and semigroups

One of the finer points of the Haskell community has been its propensity for recognizing abstract patterns in code that have well-defined, lawful representations in mathematics. A word frequently used to describe these abstractions is *algebra*, by which we mean one or more *operations* and the *set* they operate over. Over the next few chapters, we're going to be looking at some of these. Some you may have heard of, such as functor and monad. Some, such as monoid and the humble semigroup, may seem new to you. One of the things that Haskell is really good at is these algebras, and it's important to master them before we can do some of the exciting stuff that's coming.

This chapter will include:

- Algebras!
- Laws!
- Monoids!
- Semigroups!

15.2 What we mean by algebra

For some of us, talking about "an algebra" may sound somewhat foreign. So let's take a second and talk about what we're talking about when we use this phrase, at least when we're talking about Haskell.

Algebra generally refers to one of the most important fields of mathematics. In this usage, it means the study of mathematical symbols and the rules governing their manipulation. It is differentiated from arithmetic by its use of abstractions such as variables. By the use of variables, we're saying we don't care much what value will be put into that slot. We care about the rules of how to manipulate this thing without reference to its particular value.

And so, as we said above, *an algebra* refers to some operations and the set they operate over. Here again, we care less about the particulars of the values or data we're working with and more about the general rules of their use.

In Haskell, these algebras can be implemented with type classes—the type class defines the set of operations. When we talk about

operations over a set, the set is the *type* the operations are for. The instance defines how each operation will perform for a given type or set. One of those algebras we use is *monoid*. If you're a working programmer, you've probably had monoidal patterns in your code already, perhaps without realizing it.

15.3 Monoid

A monoid is a binary associative operation with an identity. This definition tells you a lot—if you're accustomed to picking apart mathematical definitions. Let us dissect this frog!

A **monoid** is a **binary associative operation** with an **identity**.

1. **Monoid**: The thing we're talking about—monoids. That'll end up being the name of our type class.
2. **Binary**, i.e., two. So, there will be two of something.
3. **Associative**—this is a property or law that must be satisfied. You've seen associativity with addition and multiplication. We'll explain it more in a moment.
4. **Operation**—so called because in mathematics, it's usually used as an infix operator. You can read this interchangeably as "function." Note that given the mention of "binary" earlier, we know that this is a function of two arguments.
5. **Identity** is one of those words in mathematics that pops up a lot. In this context, we can take this to mean there'll be some value which, when combined with any other value, will always return that other value. This can be seen most immediately with examples.

   For lists, we have a binary operator, `++`, that joins two lists together. We can also use a function, `mappend`, from the `Monoid` type class to do the same thing:

   ```
   Prelude> mappend [1, 2, 3] [4, 5, 6]
   [1,2,3,4,5,6]
   ```

   For lists, the empty list, `[]`, is the identity value:

```
mappend [1..5] [] = [1..5]
mappend [] [1..5] = [1..5]
```

We can rewrite this as a more general rule, using `mempty` from the `Monoid` type class as a generic identity value (more on this later):

```
mappend x mempty = x
mappend mempty x = x
```

In plain English, a monoid is a function that takes two arguments and follows two laws: associativity and identity. Associativity means the arguments can be regrouped (or re-parenthesized, or re-associated) in different orders and give the same result, as in addition. Identity means there exists some value such that when we pass it as an input to our function, the operation is rendered moot and the other value is returned, such as when we add zero or multiply by one. `Monoid` is the type class that generalizes these laws across types.

15.4 How Monoid is defined in Haskell

Type classes give us a way to recognize, organize, and use common functionalities and patterns across types that differ in some ways but also have things in common. So, we recognize that, although there are many types of numbers, all of them can be arguments in addition, and then we make an addition function as part of the `Num` class that all numbers implement.

The `Monoid` type class recognizes and orders a different pattern than `Num`, but the goal is similar. The pattern of `Monoid` is outlined above: types that have binary functions that let you join things together in accordance with the laws of associativity, along with an identity value that will return the other argument unmodified. This is the pattern of summation, multiplication, and list concatenation, among other things. The type class abstracts and generalizes the pattern so that you write code in terms of *any* type that can be monoidally combined.

The type class `Monoid` is defined like this:

```
class Semigroup m => Monoid m where
  mempty  :: m
  mappend :: m -> m -> m
  mconcat :: [m] -> m
  mconcat = foldr mappend mempty
```

`class Semigroup m => Monoid m where` means that we are defining a type class named `Monoid`, which *must* also have an instance of `Semigroup`.

`mappend` is how *any* two values that inhabit your type can be joined together. `mempty` is the identity value for that `mappend` operation. There are some laws that all `Monoid` instances must abide, and we'll get to those soon. Next, let's look at some examples of monoids in action!

15.5 Examples of using Monoid

The nice thing about monoids is that they are familiar; they're all over the place. The best way to understand them initially is to look at examples of some common monoidal operations and remember that this type class abstracts the *pattern* out, giving you the ability to use the operations over a larger range of types.

List

One common type with an instance of `Monoid` is the list type. Check out how monoidal operations work with lists:

```
Prelude> mappend [1, 2, 3] [4, 5, 6]
[1,2,3,4,5,6]
Prelude> mconcat [[1..3], [4..6]]
[1,2,3,4,5,6]
Prelude> let g = " goes well with garlic"
Prelude> mappend "Trout" g
"Trout goes well with garlic"
```

This should look familiar, because we've certainly seen this before:

```
Prelude> (++) [1, 2, 3] [4, 5, 6]
[1,2,3,4,5,6]
Prelude> let g = " goes well with garlic"
Prelude> (++) "Trout" g
```

```
"Trout goes well with garlic"
Prelude> foldr (++) [] [[1..3], [4..6]]
[1,2,3,4,5,6]
Prelude> foldr mappend mempty [[1..3], [4..6]]
[1,2,3,4,5,6]
```

Our old friend ++! And if we look at a possible definition of `Semigroup` and `Monoid` for lists, we can see how this all lines up:

```
instance Semigroup [a] where
  (<>) = (++)

instance Monoid [a] where
  mempty  = []
```

We didn't have to explicitly define `mappend` for our list `Monoid`. There's a default definition of `mappend` in the definition of the `Monoid` class that takes advantage of the fact that we must have already defined a `Semigroup` before we are permitted to implement the `Monoid` instance:

```
-- From base-4.11.1.0

class Semigroup a => Monoid a where
  mempty  :: a
  mappend :: a -> a -> a
  mappend = (<>)
```

For other types, the instances would be different, but the ideas behind them remain the same.

15.6 Why Integer doesn't have a Monoid

The type `Integer` does not have a `Monoid` instance. None of the numeric types do. Yet it's clear that numbers have monoidal operations, so what's up with that, Haskell?

While, in mathematics, the monoid of numbers is summation, there's no clear reason why it can't be multiplication. Both operations are monoidal (binary, associative, having an identity value), but each type should only have one unique instance for a given type class, not two (one instance for a sum, one for a product).

This won't work:

```
Prelude> x = 1 :: Integer
Prelude> y = 3 :: Integer
Prelude> mappend x y

• No instance for (Monoid Integer)
  arising from a use of ‘mappend’
• In the expression: mappend x y
  In an equation for ‘it’: it = mappend x y
```

It isn't clear if those should be added or multiplied as a `mappend` operation. It says there's no `Monoid` for those `Integer`s for that reason. You get the idea.

To resolve the conflict, we have the `Sum` and `Product` newtypes to wrap numeric values and signal which `Monoid` instance we want. These newtypes are built into the `Data.Monoid` module. While there are two possible instances of `Monoid` for numeric values, we avoid using scoping tricks and abide by the rule that type class instances are *unique* to the types they are for:

```
Prelude> mappend (Sum 1) (Sum 5)
Sum {getSum = 6}
Prelude> mappend (Product 5) (Product 5)
Product {getProduct = 25}
Prelude> mappend (Sum 4.5) (Sum 3.4)
Sum {getSum = 7.9}
```

Note that we could use it with values that aren't integral. We can use these `Monoid` newtypes for all the types that have instances of `Num`.

Integers form a monoid under summation and multiplication. We can similarly say that lists form a monoid under concatenation.

It's worth pointing out here that numbers aren't the only sets that have more than one possible monoid. Lists have more than one possible monoid, although for now we're only working with concatenation (we'll look at the other list monoid in another chapter). Several other types do, as well. We usually enforce the unique instance rule by using `newtype` to separate the different monoidal behaviors.

Why newtype?

Use of a `newtype` can be hard to justify or explain to people that don't yet have good intuitions for how Haskell code gets compiled and the representations of data used by your computer in the course of executing your programs. With that in mind, we'll do our best and offer two explanations intended for two different audiences. We will return to the topic of `newtype` in more detail later in the book.

First, there's not much semantic difference (except for circumstances involving `bottom`, explained later) between the following datatypes:

```
data Server = Server String

newtype Server' = Server' String
```

The main differences are that using `newtype` *constrains* the datatype to having a single, unary data constructor and `newtype` guarantees no additional runtime overhead in "wrapping" the original type. That is, the runtime representation of `newtype` and what it wraps are always identical—no additional "boxing up" of the data is necessary for typical products and sums.

For veteran programmers who understand pointers `newtype` is like a single-member C union that avoids creating an extra pointer but still gives you a new type constructor and data constructor so you don't mix up the *many many many* things that share a single representation.

In summary, why you might use newtype

1. To signal intent: using `newtype` makes it clear that you only intend for it to be a wrapper for the underlying type. The newtype cannot eventually grow into a more complicated sum or product type, while a normal datatype can.
2. To improve type safety: avoid mixing up many values of the same representation, such as `Text` or `Integer`.
3. To add different type class instances to a type that is otherwise unchanged representationally, such as with `Sum` and `Product`.

More on Sum and Product

There's more than one valid `Monoid` instance one can write for numbers, so we use `newtype` wrappers to distinguish which we want. If you import `Data.Monoid`, you'll see the `Sum` and `Product` newtypes:

```
Prelude> import Data.Monoid
Prelude> :info Sum
newtype Sum a = Sum {getSum :: a}
...some instances elided...
instance Num a => Monoid (Sum a)

Prelude> :info Product
newtype Product a =
   Product {getProduct :: a}
...some instances elided...
instance Num a => Monoid (Product a)
```

The instances say that we can use `Sum` or `Product` values as a `Monoid` as long as they contain numeric values. We can prove this is the case for ourselves. We're going to be using the infix operator for `mappend` in these examples. It has the same type and does the same thing but saves some characters and will make these examples a bit cleaner:

```
Prelude Data.Monoid> :t (<>)
(<>) :: Monoid m => m -> m -> m

Prelude> Sum "Frank" <> Sum "Herbert"

• No instance for (Num [Char]) ...
```

This example doesn't work, because the `a` in `Sum a` is type `String`, which is not an instance of `Num`.

`Sum` and `Product` do what you'd expect with a bit of syntactic surprise:

```
Prelude Data.Monoid> (Sum 8) <> (Sum 9)
Sum {getSum = 17}

Prelude Data.Monoid> mappend mempty Sum 9
Sum {getSum = 9}
```

But `mappend` joins two things, so you can't do this:

```
Prelude> mappend (Sum 8) (Sum 9) (Sum 10)
```

You'll get a big error message, including this line:

```
• The function ‘mappend’ is applied to
  three arguments,
  but its type ‘Sum Integer -> Sum Integer
    -> Sum Integer’
  has only two
```

So, that's easy enough to fix by nesting:

```
Prelude> x = Sum 1
Prelude> y = Sum 2
Prelude> z = Sum 3
Prelude> mappend x (mappend y z)
Sum {getSum = 6}
```

Or, somewhat less tediously, by infixing the `mappend` function:

```
Prelude> Sum 1 <> Sum 1 <> Sum 1
Sum {getSum = 3}
```

Or, you could also put your `Sum` values in a list and use `mconcat`:

```
Prelude> mconcat [Sum 8, Sum 9, Sum 10]
Sum {getSum = 27}
```

Due to the special syntax of `Sum` and `Product`, we also have built-in record field accessors we can use to unwrap their values:

```
Prelude> x = Sum 1
Prelude> getSum $ mappend x x
2
Prelude> y = Product 5
Prelude> getProduct $ mappend y y
25
Prelude> a = Sum 5
Prelude> b = Sum 6
Prelude> c = Sum 7
```

```
Prelude> getSum $ mconcat [a, b, c]
18
```

`Product` is similar to `Sum` but for multiplication.

15.7 Why bother?

Because monoids are common, and they're a nice abstraction to work with when you have multiple monoidal things running around in a project. Knowing what a monoid is can help you to recognize when you encounter the pattern. Further, having principled laws for it means you know you can combine monoidal operations safely. When we say something *is a monoid* or can be described as *monoidal*, we mean you can define at least one law-abiding `Monoid` instance for it.

A common use of monoids is to structure and describe common modes of processing data. Sometimes, this is to describe an API for incrementally processing a large dataset, sometimes to describe guarantees needed to roll up aggregations (think summation) in a parallel, concurrent, or distributed processing framework.

One example of where things like the identity can be useful is if you want to write a generic library for doing work in parallel. You could choose to describe your work as being like a tree, with each unit of work being a leaf. From there, you can partition the tree into as many chunks as are necessary to saturate the number of processor cores or entire computers you want to devote to the work. The problem is, if we have a pair-wise operation and we need to combine an odd number of leaves, how do we even out the count?

One straightforward way could be to simply provide `mempty` (the identity value) to the odd leaves out, so we get the same result and pass it up to the next layer of aggregation!

A variant of monoid that provides more guarantees is the Abelian or commutative monoid. Commutativity can be particularly helpful when doing concurrent or distributed processing of data, because it means the intermediate results being computed in a different order won't change the eventual answer.

Monoids are even strongly associated with the concept of *folding* or catamorphism—something we do *all the time* in Haskell. You'll see this more explicitly in Chapter 20, but here's a taste:

```
Prelude> xs = [2, 4, 6] :: [Product Int]
Prelude> foldr mappend mempty xs
Product {getProduct = 48}

Prelude> ys = [2, 4, 6] :: [Sum Int]
Prelude> foldr mappend mempty ys
Sum {getSum = 12}

Prelude> strList = ["blah", "woot"]
Prelude> foldr mappend mempty strList
"blahwoot"
```

You'll see monoidal structure come up when we explain `Applicative` and `Monad`, as well.

15.8 Laws

We'll get to those laws in a moment. First, heed our little *cri de coeur* about why you should care about mathematical laws:

Laws circumscribe what constitutes a valid instance or concrete instance of the *algebra*, or set of operations, we're working with. We care about the laws a `Monoid` instance must adhere to, because we want our programs to be correct wherever possible. Proofs are programs, and programs are proofs. We care about programs that compose well, that are easy to understand, and which have predictable behavior. To that end, we should steal *prolifically* from mathematics.

Algebras are *defined* by their laws and are useful principally *for* their laws. Laws make up what algebras *are*.

Among other things, laws provide us guarantees that let us build on solid foundations. Those guarantees give us predictable composition (or combination) of programs. Without the ability to safely combine programs, everything must be written from scratch, and nothing could be reused. The physical world has enjoyed the useful properties of stone stacked up on top of stone since the Great Pyramid of Giza was built in the pharaoh Sneferu's reign in 2,600 BC. Similarly, if we want to be able to stack up functions scalably, they need to obey laws. Stones don't evaporate into thin air or explode violently. It'd be nice if our programs were similarly trustworthy.

There are more possible laws we can require for an algebra than associativity or identity, but these are simple examples we are starting with for now, partly because `Monoid` is a good place to start with algebras-as-type-classes. We'll see examples of more later.

`Monoid` instances must abide by the following laws:

```
-- left identity
mappend mempty x = x

-- right identity
mappend x mempty = x

-- associativity
mappend x (mappend y z) =
  mappend (mappend x y) z

mconcat = foldr mappend mempty
```

Any laws that apply to `mappend` will also apply to the `<>` method defined by `Semigroup`. More on that later in this chapter. Here is how the identity law looks in practice:

```
Prelude> import Data.Monoid

-- left identity
Prelude> mappend mempty (Sum 1)
Sum {getSum = 1}

-- right identity
Prelude> mappend (Sum 1) mempty
Sum {getSum = 1}
```

We can demonstrate associativity using the infix operator `<>` from the `Semigroup` class. Remember that `mappend` and `<>` should be identical in behavior. Note the parenthesization in these two examples:

```
Prelude> :t (<>)
(<>) :: Semigroup m => m -> m -> m
```

Associativity:

```
Prelude> (Sum 1) <> (Sum 2 <> Sum 3)
Sum {getSum = 6}

Prelude> (Sum 1 <> Sum 2) <> (Sum 3)
Sum {getSum = 6}
```

And `mconcat` should have the same result as `foldr mappend mempty`:

```
Prelude> xs = [Sum 1, Sum 2, Sum 3]
Prelude> mconcat xs
Sum {getSum = 6}

Prelude> foldr mappend mempty xs
Sum {getSum = 6}
```

Now, let's see all of that again, but using the `Monoid` of lists, for which `mempty` is `[]` and `mappend` is `++`:

```
-- left identity
Prelude> mappend mempty [1, 2, 3]
[1,2,3]

-- right identity
Prelude> mappend [1, 2, 3] mempty
[1,2,3]

-- associativity
Prelude> [1] <> ([2] <> [3])
[1,2,3]

Prelude> ([1] <> [2]) <> [3]
[1,2,3]

-- mconcat ~ foldr mappend mempty
Prelude> mconcat [[1], [2], [3]]
[1,2,3]

Prelude> foldr mappend mempty [[1], [2], [3]]
[1,2,3]
```

```
Prelude> concat [[1], [2], [3]]
[1,2,3]
```

The important part here is that you have these guarantees even when you don't know *which monoid* you'll be working with.

15.9 Different instance, same representation

`Monoid` is somewhat different from other type classes in Haskell, in that many datatypes have more than one valid monoid. We saw that for numbers, both addition and multiplication are sensible monoids with different behaviors. When we have more than one potential implementation for `Monoid` for a datatype, it's most convenient to use newtypes to distinguish them, as we did with `Sum` and `Product`.

Addition is a classic appending operation, as is list concatenation. Referring to multiplication as an appending operation may also seem intuitive enough, as it still follows the basic pattern of combining two values of one type into one value.

But for other datatypes, the meaning of append is less clear. In these cases, the monoidal operation is less about combining the values and more about finding a summary value for the set. We mentioned above that monoids are important to folding and catamorphisms, more generally. Appending is perhaps best thought of not as a way of combining values in the way that addition or list concatenation does, but as a way to condense any set of values to a summary value. We'll start by looking at the `Monoid` instances for `Bool` to see what we mean.

Boolean values have two possible monoids—a monoid of conjunction and a monoid of disjunction. As we do with numbers, we use newtypes to distinguish the two instances. `All` and `Any` are the newtypes for `Bool`'s monoids:

```
Prelude> import Data.Monoid

Prelude> All True <> All True
All {getAll = True}
Prelude> All True <> All False
All {getAll = False}
```

```
Prelude> Any True <> Any False
Any {getAny = True}
Prelude> Any False <> Any False
Any {getAny = False}
```

`All` represents Boolean conjunction: it returns a `True` if and only if all values it is "appending" are `True`. `Any` is the monoid of Boolean disjunction: it returns a `True` if any value is `True`. There is a sense in which it feels strange to think of this as a combining operation. Remember, this is about condensing or reducing the values.

The `Maybe` type has more than two possible `Monoids`. We'll look at each in turn, but the two that have an obvious relationship are `First` and `Last`. They are like Boolean disjunction but with explicit preference for the leftmost or rightmost success in a series of `Maybe` values. We have to choose, because with `Bool`, all you know is `True` or `False`—it doesn't matter where your `True` or `False` values occur. With `Maybe`, however, you need to make a decision as to which `Just` value you'll return, if there are multiple successes. `First` and `Last` encode these different possibilities.

`First` returns the first or leftmost non-`Nothing` value:

```
Prelude> x = First (Just 1)
Prelude> y = First (Just 2)
Prelude> x `mappend` y
First {getFirst = Just 1}
```

`Last` returns the last or rightmost non-`Nothing` value:

```
Prelude> x = Last (Just 1)
Prelude> y = Last (Just 2)
Prelude> x `mappend` y
Last {getLast = Just 2}
```

Both will succeed in returning *something*, in spite of `Nothing` values, as long as there's at least one `Just`:

```
Prelude> last = Last (Just 2)
Prelude> Last Nothing `mappend` last
Last {getLast = Just 2}
```

```
Prelude> first = First (Just 2)
Prelude> First Nothing `mappend` first
First {getFirst = Just 2}
```

Neither can, for obvious reasons, return anything if all values are `Nothing`:

```
Prelude> noFirst = First Nothing
Prelude> noFirst `mappend` noFirst
First {getFirst = Nothing}

Prelude> noLast = Last Nothing
Prelude> noLast `mappend` noLast
Last {getLast = Nothing}
```

To maintain the unique pairing of type and type class instance, newtypes are used for all of those, the same as we saw with `Sum` and `Product`.

Let's look next at the third variety of `Maybe Monoid`.

15.10 Reusing algebras by asking for algebras

We alluded to there being more possible `Monoids` for `Maybe` than just `First` and `Last`. Let's write that other `Monoid` instance. We will now be concerned not with choosing one value out of a set of values but of combining the `a` values contained within the `Maybe a` type.

First, try to notice a pattern:

```
instance Monoid b => Monoid (a -> b)
instance (Monoid a, Monoid b)
      => Monoid (a, b)
instance (Monoid a, Monoid b, Monoid c)
      => Monoid (a, b, c)
```

What these `Monoids` have in common is that they are giving you a new `Monoid` for a larger type by reusing the `Monoid` instances of types that represent components of the larger type.

This obligation to ask for a `Monoid` for an encapsulated type (such as the `a` in `Maybe a`) exists even when not all possible values of the larger type contain the value of the type argument. For example,

`Nothing` does not contain the `a` we're trying to get a `Monoid` for, but `Just a` does, so not all possible `Maybe` values contain the `a` type argument. For a `Maybe Monoid` that will have a `mappend` operation for the `a` values, we need a `Monoid` for whatever type `a` is. `Monoid` types like `First` and `Last` wrap the `Maybe a` but do not require a `Monoid` for the `a` value itself, because they don't `mappend` the `a` values or provide a `mempty` for them.

If you do have a datatype that has a type argument that does not appear anywhere in the terms (a phantom type), the type checker does not demand that you have a `Monoid` instance for that argument. For example:

```
data Booly a =
    False'
  | True'
  deriving (Eq, Show)
```

We'll need to define a `Semigroup` before we define a `Monoid`:

```
-- conjunction
instance Semigroup (Booly a) where
  (<>) False' _ = False'
  (<>) _ False' = False'
  (<>) True' True' = True'

instance Monoid (Booly a) where
  mempty = True'
```

We didn't need a `Monoid` constraint for `a`, because we're never combining `a` values (we can't, as none exist), and we're never asking for a `mempty` of type `a`. This is the fundamental reason we don't need the constraint, but it can happen that we don't do this even when the type *does* occur in the datatype.

Exercise: Optional Monoid

Write the `Monoid` instance for our `Maybe` type, renamed to `Optional`:

```
data Optional a =
    Nada
  | Only a
  deriving (Eq, Show)

instance Monoid a
      => Monoid (Optional a) where
  mempty = undefined
  mappend = undefined
```

Expected output:

```
Prelude> onlySum = Only (Sum 1)
Prelude> onlySum `mappend` onlySum
Only (Sum {getSum = 2})

Prelude> onlyFour = Only (Product 4)
Prelude> onlyTwo = Only (Product 2)
Prelude> onlyFour `mappend` onlyTwo
Only (Product {getProduct = 8})

Prelude> Only (Sum 1) `mappend` Nada
Only (Sum {getSum = 1})

Prelude> Only [1] `mappend` Nada
Only [1]

Prelude> Nada `mappend` Only (Sum 1)
Only (Sum {getSum = 1})
```

Associativity

This will be mostly a review, but we want to be specific about associativity. Associativity says that you can associate, or group, the arguments of your operation differently, and the result will be the same.

Let's review examples of some operations that can be *re-associated*:

```
Prelude> (1 + 9001) + 9001
```

```
18003
Prelude> 1 + (9001 + 9001)
18003
Prelude> (7 * 8) * 3
168
Prelude> 7 * (8 * 3)
168
```

And some that cannot have the parentheses re-associated without changing the result:

```
Prelude> (1 - 10) - 100
-109
Prelude> 1 - (10 - 100)
91
```

This is *not* as strong a property as an operation that commutes or is *commutative*. Commutative means you can reorder the arguments and still get the same result. Addition and multiplication are commutative, but ++ for the list type is only associative.

Let's demonstrate this by writing a mildly evil version of addition that flips the order of its arguments:

```
Prelude> evilPlus = flip (+)
Prelude> 76 + 67
143
Prelude> 76 `evilPlus` 67
143
```

We have some evidence, but not *proof*, that + commutes.

However, we can't do the same with ++:

```
Prelude> evilPlusPlus = flip (++)
Prelude> oneList = [1..3]
Prelude> otherList = [4..6]

Prelude> oneList ++ otherList
[1,2,3,4,5,6]

Prelude> oneList `evilPlusPlus` otherList
[4,5,6,1,2,3]
```

This serves as a proof by counterexample that ++ does *not* commute. It doesn't matter if it commutes for all other inputs; that it doesn't commute for one of them means the *law of commutativity* does not hold here.

Commutativity is a useful property and can be helpful in circumstances when you might need to be able to reorder evaluation of your data for efficiency purposes without needing to worry about the result changing. Distributed systems use commutative monoids in designing and thinking about constraints, which are monoids that guarantee their operation commutes.

But, for our purposes, `Monoid` abides by the law of associativity but not the law of commutativity, even though some monoidal operations (addition and multiplication) are commutative.

Identity

An identity is a value with a special relationship with an operation: it turns the operation into the identity function. There are no identities without operations. The concept is defined in terms of its relationship with a given operation. If you've done grade school arithmetic, you've already seen some identities:

```
Prelude> 1 + 0
1
Prelude> 521 + 0
521
Prelude> 1 * 1
1
Prelude> 521 * 1
521
```

Zero is the identity value for addition, while one is the identity value for multiplication. As we said, it doesn't make sense to talk about zero and one as identity values outside the context of those operations. That is, zero is definitely not the identity value for other operations. We can check this property with a simple equality test, as well:

```
Prelude> myList = [1..424242]
```

```
-- 0 serves as identity for addition
Prelude> map (+0) myList == myList
True
-- but not for multiplication
Prelude> map (*0) myList == myList
False

-- 1 serves as identity for multiplication
Prelude> map (*1) myList == myList
True

-- but not for addition
Prelude> map (+1) myList == myList
False
```

This is the other law for `Monoid`: the binary operation must be associative, *and* it must have a sensible identity value.

The problem of orphan instances

We've said both in this chapter and in the earlier chapter devoted to type classes that type classes have unique pairings of the class and the instance for a particular type.

We do sometimes end up with multiple instances for a single type when orphan instances are written. But writing orphan instances should be avoided *at all costs*. If you get an orphan instance warning from GHC, *fix it*.

An orphan instance is when an instance is defined for a datatype and type class but not in the same module as either the declaration of the type class or the datatype. If you don't own the type class or the datatype, newtype it!

If you want an orphan instance so that you can have multiple instances for the same type, you still want to use `newtype`. We saw this earlier with `Sum` and `Product`, which let us have two different `Monoid` instances for numbers without resorting to orphans or messing up type class instance uniqueness.

Let's see an example of an orphan instance and how to fix it. First, make a project directory and change into that directory:

```
$ mkdir orphan-instance && cd orphan-instance
```

Then, we're going to make a couple of files, one module in each:

```
module Listy where

newtype Listy a =
  Listy [a]
  deriving (Eq, Show)

instance Semigroup (Listy a) where
  (<>) (Listy l) (Listy l') =
    Listy $ mappend l l'
```

```
module ListyInstances where

import Data.Monoid
import Listy

instance Semigroup (Listy a) where
  (<>) (Listy l) (Listy l') =
    Listy $ mappend l l'


instance Monoid (Listy a) where
  mempty = Listy []
```

Your directory should look like this:

```
$ tree
.
├── Listy.hs
└── ListyInstances.hs
```

Then, to build `ListyInstances` such that it can see `Listy`, we must use the `-I` flag to include the current directory and make modules within it discoverable. The `.` after the `I` is how we say "this directory" in Unix-alikes. If you succeed, you should see something like the following:

```
$ ghc -I. --make ListyInstances.hs
[2 of 2] Compiling ListyInstances
```

Note that the only output will be an object file, the result of compiling a module that can be reused as a library by Haskell code, because we didn't define a `main` suitable for producing an executable. We're only using this approach to build this so that we can avoid the hassle of initializing a project (via `stack new` or similar). For *anything* more complicated or long-lived than this, use a dependency and build management tool like Cabal (if you're using Stack, you're also using Cabal).

Now, to provide one example of why orphan instances are problematic. If we copy our `Monoid` instance from `ListyInstances` into `Listy`, then rebuild `ListyInstances`, we'll get the following error:

```
$ ghc -I. --make ListyInstances.hs
[1 of 2] Compiling Listy
[2 of 2] Compiling ListyInstances

Listy.hs:7:10: error:
    Duplicate instance declarations:
      instance [safe] Semigroup (Listy a)
      instance Semigroup (Listy a)
  |
7 | instance Semigroup (Listy a) where
```

These conflicting instance declarations could happen to anybody who uses the previous version of our code. And that's a problem.

Orphan instances are *still* a problem even if duplicate instances aren't both imported into a module, because it means your type class methods will start behaving differently depending on what modules are imported, which breaks the fundamental assumptions and niceties of type classes.

There are a few solutions for addressing orphan instances:

1. You defined the type but not the type class? Put the instance in the same module as the type, so that the type cannot be imported without its instances.

2. You defined the type class but not the type? Put the instance in the same module as the type class definition, so that the type class cannot be imported without its instances.

3. Neither the type nor the type class are yours? Define your own newtype wrapping the original type and now you've got a type that "belongs" to you for which you can rightly define type class instances. There are means of making this less annoying, which we'll discuss later.

These restrictions must be maintained in order for us to reap the full benefit of type classes along with the reasoning properties that are associated with them. A type *must* have a unique (singular) implementation of a type class in scope, and avoiding orphan instances is how we prevent conflicting instances. Be aware, however, that avoidance of orphan instances is more strictly adhered to among library authors rather than application developers, although it's no less important in applications.

15.11 Madness

You may have seen mad libs before. The idea is to take a template of phrases, fill them in with blindly selected categories of words, and see if saying the final version is amusing.

Using a lightly edited example from the Wikipedia article on Mad Libs:

```
"___________! he said ______ as he
 exclamation          adverb
jumped into his car ____ and drove
                    noun
off with his _________ wife."
             adjective
```

We can make this into a function, like the following:

```
import Data.Monoid

type Adjective = String
type Adverb = String
type Noun = String
type Exclamation = String
```

```
madlibbin' :: Exclamation
           -> Adverb
           -> Noun
           -> Adjective
           -> String
madlibbin' e adv noun adj =
  e   <> "! he said " <>
  adv <> " as he jumped into his car " <>
  noun <> " and drove off with his " <>
  adj <> " wife."
```

Now, you're going to refactor this code a bit! Rewrite it using `mconcat`:

```
madlibbinBetter' :: Exclamation
                 -> Adverb
                 -> Noun
                 -> Adjective
                 -> String
madlibbinBetter' e adv noun adj = undefined
```

15.12 Better living through QuickCheck

Proving laws can be tedious, especially if the code we're checking is in the middle of changing frequently. Accordingly, having a cheap way to get a sense of whether or not the laws are *likely* to be obeyed by an instance is pretty useful. `QuickCheck` happens to be an excellent way to accomplish this.

Validating associativity with QuickCheck

You can check the associativity of some simple arithmetic expressions by asserting equality between two versions with different parenthesization and checking them in the REPL:

```
-- we're saying these are the same because
-- + and * are associative

1 + (2 + 3) == (1 + 2) + 3

4 * (5 * 6) == (4 * 5) * 6
```

This doesn't tell us that associativity holds for *any* inputs to `+` and `*`, though, and that is what we want to test. Our old friend from the lambda calculus—abstraction!—suffices for this:

```
\ a b c -> a + (b + c) == (a + b) + c

\ a b c -> a * (b * c) == (a * b) * c
```

But our arguments aren't the only things we can abstract. What if we want to talk about the *abstract* property of associativity for some given function `f`?

```
\ f a b c ->
  f a (f b c) == f (f a b) c

\ (<>) a b c ->
  a <> (b <> c) == (a <> b) <> c
```

Surprise! You can bind infix names for function arguments:

```
asc :: Eq a
    => (a -> a -> a)
    -> a -> a -> a
    -> Bool
asc (<>) a b c =
  a <> (b <> c) == (a <> b) <> c
```

Now, how do we turn this function into something we can property test with `QuickCheck`? The quickest and easiest way would probably look something like the following:

```
import Data.Monoid
import Test.QuickCheck

monoidAssoc :: (Eq m, Monoid m)
            => m -> m -> m -> Bool
monoidAssoc a b c =
  (a <> (b <> c)) == ((a <> b) <> c)
```

We have to declare the types for the function in order to run the tests, so that `QuickCheck` knows what types of data to generate.

We can now use this to check the associativity of functions:

```
Prelude> type S = String
Prelude> type B = Bool
Prelude> type MA = S -> S -> S -> B
Prelude> quickCheck (monoidAssoc :: MA)
+++ OK, passed 100 tests.
```

The `quickCheck` function uses the `Arbitrary` type class to provide the randomly generated inputs for testing the function. Although it's common to do so, we may not want to rely on an `Arbitrary` instance existing for the type of our inputs, for one of a few reasons. It may be that we need a generator for a type that doesn't belong to us, so we'd rather not make an orphan instance. Or it could be a type that already has an `Arbitrary` instance, but we want to run tests with a different random distribution of values, or to make sure we check certain special edge cases in addition to the random values.

You want to be careful to assert types, so that `QuickCheck` knows from which `Arbitrary` instance to get random values for testing. You can use `verboseCheck` to see what values are tested. If you try running the check verbosely and without asserting a type for the arguments:

```
Prelude> verboseCheck monoidAssoc
Passed:
()
()
()

(repeated 100 times)
```

This is GHCi's type-defaulting biting you, as we saw back in Chapter 14, on testing. GHCi has slightly more aggressive type-defaulting, which can be handy in an interactive session when you want to fire off some code and have your REPL pick a winner for the type classes it doesn't know how to dispatch. Compiled in a source file, GHC would complain about an ambiguous type.

Testing left and right identity

Following on from what we did with associativity, we can also use `QuickCheck` to test left and right identity:

```
monoidLeftIdentity :: (Eq m, Monoid m)
                   => m
                   -> Bool
monoidLeftIdentity a = (mempty <> a) == a

monoidRightIdentity :: (Eq m, Monoid m)
                    => m
                    -> Bool
monoidRightIdentity a = (a <> mempty) == a
```

Then, running these properties against a `Monoid`:

```
Prelude> mli = monoidLeftIdentity
Prelude> mri = monoidRightIdentity
Prelude> quickCheck (mli :: String -> Bool)
+++ OK, passed 100 tests.

Prelude> quickCheck (mri :: String -> Bool)
+++ OK, passed 100 tests.
```

Testing QuickCheck's patience

Let us see an example of `QuickCheck` catching us out for having an invalid `Monoid`. We're going to demonstrate why a `Bool` `Monoid` can't have `False` as the identity, always returning the value `False`, and still be a valid `Monoid`. Associative, left identity, and right identity properties have been elided from the following example. Add them:

```
import Control.Monad
import Data.Monoid
import Test.QuickCheck

data Bull =
    Fools
  | Twoo
  deriving (Eq, Show)

instance Arbitrary Bull where
  arbitrary =
    frequency [ (1, return Fools)
              , (1, return Twoo) ]

instance Semigroup Bull where
  (<>) _ _ = Fools

instance Monoid Bull where
  mempty = Fools

type BullMappend =
  Bull -> Bull -> Bull -> Bool

main :: IO ()
main = do
  let ma = monoidAssoc
      mli = monoidLeftIdentity
      mri = monoidRightIdentity
  quickCheck (ma :: BullMappend)
  quickCheck (mli :: Bull -> Bool)
  quickCheck (mri :: Bull -> Bool)
```

If you load this up in GHCi and run `main`, you'll get the following output:

```
Prelude> main
+++ OK, passed 100 tests.
*** Failed! Falsifiable (after 1 test):
Twoo
```

```
*** Failed! Falsifiable (after 1 test):
Twoo
```

So this not-actually-a-`Monoid` for `Bool` turns out to pass associativity but fail on the right and left identity checks. To see why, let's line up the laws against what our `mempty` and `mappend` are:

```
-- how the instance is defined
mempty = Fools
-- remember, mappend == <>
mappend _ _ = Fools

-- identity laws
mappend mempty x = x
mappend x mempty = x
```

Does it obey the laws?

```
-- because of how mappend is defined
mappend mempty x = Fools
mappend x mempty = Fools
```

`Fools` is not `x`, so this fails the identity laws.

It's fine if your identity value is `Fools`, but if your `mappend` always returns the identity, then it's not an identity. It's not behaving like a zero, as you're not even checking if either argument is `Fools` before returning `Fools`. It's a black hole that spits out one value, which is senseless. For an example of what is meant by zero, consider multiplication, which has an identity *and* a zero:

```
-- Thus why the mempty for Sum is 0
0 + x == x
x + 0 == x

-- Thus why the mempty for Product is 1
1 * x == x
x * 1 == x

-- Thus why the mempty for
-- Product is *not* 0
0 * x == 0
x * 0 == 0
```

Using `QuickCheck` can be a great way to cheaply and easily sanity check the validity of your instances against their laws. You'll see more of this.

Exercise: Maybe another Monoid

Write a `Monoid` instance for a `Maybe` type that doesn't require a `Monoid` for the contents. Reuse the `Monoid` law `QuickCheck` properties, and use them to validate the instance.

Don't forget to write an `Arbitrary` instance for `First'`. We won't always stub that out explicitly for you. We suggest learning how to use the `frequency` function from `QuickCheck` for the instance of `First'`:

```
newtype First' a =
  First' { getFirst' :: Optional a }
  deriving (Eq, Show)

instance Semigroup (First' a) where
  (<>) = undefined

instance Monoid (First' a) where
  mempty = undefined

firstMappend :: First' a
             -> First' a
             -> First' a
firstMappend = mappend

type FirstMappend =
     First' String
  -> First' String
  -> First' String
  -> Bool

type FstId =
  First' String -> Bool
```

```
main :: IO ()
main = do
  quickCheck (monoidAssoc :: FirstMappend)
  quickCheck (monoidLeftIdentity :: FstId)
  quickCheck (monoidRightIdentity :: FstId)
```

Our expected output demonstrates a different `Monoid` for `Optional`/`Maybe`, which is getting the first success and holding onto it, where any exist. This could be seen, with a bit of hand-waving, as being a disjunctive ("or") `Monoid` instance:

```
Prelude> onlyOne = First' (Only 1)
Prelude> onlyTwo = First' (Only 2)
Prelude> nada = First' Nada
Prelude> onlyOne `mappend` nada
First' {getFirst' = Only 1}
Prelude> nada `mappend` nada
First' {getFirst' = Nada}
Prelude> nada `mappend` onlyTwo
First' {getFirst' = Only 2}
Prelude> onlyOne `mappend` onlyTwo
First' {getFirst' = Only 1}
```

15.13 Semigroup

Mathematicians play with algebras like that creepy kid you knew in grade school who would pull legs off of insects. Sometimes, they glue legs onto insects too, but in the case where we're going from `Monoid` to `Semigroup`, we're pulling a leg off. In this case, the leg is our identity. To get from a monoid to a semigroup, we simply no longer furnish nor require an identity. The core operation remains binary and associative.

With this, our definition of `Semigroup` is:

```
class Semigroup a where
  (<>) :: a -> a -> a
```

And we're left with one law:

```
(a <> b) <> c = a <> (b <> c)
```

`Semigroup` still provides a binary associative operation, one that typically joins two things together (as in concatenation or summation), but it doesn't have an identity value. In that sense, it's a weaker algebra.

NonEmpty, a useful datatype

One useful datatype that can't have a `Monoid` instance but does have a `Semigroup` instance is the `NonEmpty` list type. It is a list datatype that can never be an empty list:

```
-- from Data.List.NonEmpty
data NonEmpty a = a :| [a]
```

Here `:|` is an infix data constructor that takes two (type) arguments. It's a product of `a` and `[a]`. It guarantees that we always have *at least* one value of type `a`, which `[a]` does *not* guarantee, as any list might be empty.

Note that although `:|` is not alphanumeric, as most of the other data constructors you're used to seeing are, it is a name for an infix data constructor. Data constructors with only non-alphanumeric symbols and that begin with a colon are infix by default; those with alphanumeric names are prefix by default:

```
-- Prefix, works.
data P =
  Prefix Int String

-- Infix, works.
data Q =
  Int :!!: String
```

Since that data constructor is symbolic rather than alphanumeric, it can't be used as a prefix:

```
data R =
  :!!: Int String
```

Using it as a prefix will cause a syntax error:

```
parse error on input ‘:!!:’
```

On the other hand, an alphanumeric data constructor can't be used as an infix:

```
data S =
  Int Prefix String
```

It will cause another error:

```
Not in scope: type constructor
  or class ‘Prefix’
A data constructor of that name
  is in scope;
did you mean DataKinds?
```

Let's return to the main point, which is `NonEmpty`. Because `NonEmpty` is a product of two arguments, we could've also written it as:

```
newtype NonEmpty a =
  NonEmpty (a, [a])
  deriving (Eq, Ord, Show)
```

We can't write a `Monoid` for `NonEmpty`, because it has no identity value by design! There is no empty list to serve as an identity for any operation over a `NonEmpty` list, yet there is still a binary associative operation: two `NonEmpty` lists can still be concatenated. A type with a canonical binary associative operation but no identity value is a natural fit for `Semigroup`. Here is a brief example of using `NonEmpty` with the semigroup `mappend` (`Semigroup` and `NonEmpty` are both in `base` but not in `Prelude`):

```
Prelude> import Data.List.NonEmpty as N
Prelude N> import Data.Semigroup as S
Prelude N S> 1 :| [2, 3]
1 :| [2,3]
Prelude N S> :t 1 :| [2, 3]
1 :| [2, 3] :: Num a => NonEmpty a
Prelude N S> :t (<>)
(<>) :: Semigroup a => a -> a -> a

Prelude N S> let xs = 1 :| [2, 3]
```

```
Prelude N S> let ys = 4 :| [5, 6]
Prelude N S> xs <> ys
1 :| [2,3,4,5,6]
Prelude N S> N.head xs
1
Prelude N S> N.length (xs <> ys)
6
```

Beyond this, you use `NonEmpty` as you would a list, but what you've gained is being explicit that having zero values is not valid for your use-case. The datatype helps you enforce this constraint by not letting you construct a `NonEmpty` unless you have at least one value.

15.14 Strength can be weakness

When Haskellers talk about the *strength* of an algebra, they usually mean the number of operations it provides, which in turn expands what you can do with any given instance of that algebra without needing to know specifically what type you are working with.

The reason we cannot and do not want to make all of our algebras as big as possible is that there are datatypes that are very useful representationally, but which do not have the ability to satisfy everything in a larger algebra that could work fine if you removed an operation or law. This becomes a serious problem if `NonEmpty` is the right datatype for something in the domain you're representing. If you're an experienced programmer, think carefully. How many times have you meant for a list to never be empty? To guarantee this and make the types more informative, we use types like `NonEmpty`.

The problem is that `NonEmpty` has no identity value for the combining operation (`mappend`) in `Monoid`. So, we keep the associativity but drop the identity value and its laws of left and right identity. This is what introduces the need for and idea of `Semigroup` from a datatype.

The most obvious way to see that a monoid is *stronger* than a semigroup is to observe that it has a strict superset of the operations and laws that `Semigroup` provides. Anything that is a monoid is by definition *also* a semigroup:

```
class Semigroup a => Monoid a where
  ...
```

Earlier, we reasoned about the inverse relationship between the flexibility of your types and terms. The more information you have about the type of an expression, the fewer types that your terms can satisfy. We can see this relationship, between the number of operations—and the laws an algebra demands—and the number of datatypes that can provide a law-abiding instance of that algebra.

In the following example, `a` can be anything in the universe, but there are no operations over it—we can only return the same value.

```
id :: a -> a
```

- Number of types: infinite—universally quantified, so it can be any type the expression applying the function wants.
- Number of operations: one, if you can call it an operation, referencing the value that is passed.

With the `inc` function, `a` now has all the operations from `Num`, which lets us do more. But that also means there are now a finite set of types that can satisfy the `Num` constraint, rather than being strictly any type in the universe:

```
inc :: Num a => a -> a
```

- Number of types: anything that implements `Num`. Zero to many.
- Number of operations: seven methods in `Num`.

In the next example, we know it's an `Int`, which gives us many more operations than just a `Num` instance:

```
somethingInt :: Int -> Int
```

- Number of types: one—`Int`.
- Number of operations: considerably more than seven. In addition to `Num`, `Int` has instances of `Bounded`, `Enum`, `Eq`, `Integral`, `Ord`, `Read`, `Real`, and `Show`. On top of that, you can write arbitrary functions that pattern match on concrete types and return arbitrary values in the same type as the result. Polymorphism isn't only useful for reusing code; it's also useful for *expressing intent through parametricity*, so that people reading the code know what we are trying to accomplish.

When `Monoid` is too strong or more than we need, we can use `Semigroup`. If you're wondering what's weaker than `Semigroup`, the usual next step is removing the associativity requirement, giving you a *magma*. It's not likely to come up in day to day Haskell, but you can sound cool at programming conferences for knowing what's weaker than a semigroup, so pocket that one for the pub.

15.15 Chapter exercises

Semigroup exercises

Given a datatype, implement the `Semigroup` instance. Add `Semigroup` constraints to type variables where needed. Use the `Semigroup` class from `base` or write your own. When we use `<>`, we mean the infix `mappend` operation from the `Semigroup` type class.

Note We're not always going to derive every instance you may want or need in the datatypes we provide for exercises. We expect you to know what you need and to take care of it yourself by this point.

1. Validate *all* of your instances with `QuickCheck`. Since the only law is associativity, that's the only property you need to reuse. Keep in mind that you'll potentially need to import the modules for `Monoid` and `Semigroup` and to avoid naming conflicts for `<>`, depending on your version of GHC:

```
data Trivial = Trivial deriving (Eq, Show)

instance Semigroup Trivial where
  _ <> _ = undefined

instance Arbitrary Trivial where
  arbitrary = return Trivial

semigroupAssoc :: (Eq m, Semigroup m)
               => m -> m -> m -> Bool
semigroupAssoc a b c =
  (a <> (b <> c)) == ((a <> b) <> c)

type TrivAssoc =
  Trivial -> Trivial -> Trivial -> Bool
```

```
main :: IO ()
main =
  quickCheck (semigroupAssoc :: TrivAssoc)
```

2. ```
   newtype Identity a = Identity a
   ```

3. ```
   data Two a b = Two a b
   ```

   Hint: Ask for another `Semigroup` instance.

4. ```
   data Three a b c = Three a b c
   ```

5. ```
   data Four a b c d = Four a b c d
   ```

6. ```
   newtype BoolConj =
     BoolConj Bool
   ```

   What it should do:

   ```
   Prelude> (BoolConj True) <> (BoolConj True)
   BoolConj True
   Prelude> (BoolConj True) <> (BoolConj False)
   BoolConj False
   ```

7. ```
   newtype BoolDisj =
     BoolDisj Bool
   ```

   What it should do:

   ```
   Prelude> (BoolDisj True) <> (BoolDisj True)
   BoolDisj True
   Prelude> (BoolDisj True) <> (BoolDisj False)
   BoolDisj True
   ```

8. ```
   data Or a b =
       Fst a
     | Snd b
   ```

   The `Semigroup` for `Or` should have the following behavior. We can think of it as having a "sticky" `Snd` value, whereby it'll hold onto the first `Snd` value when and if one is passed as an argument. This is similar to the `First' Monoid` you wrote earlier:
   ```

```
Prelude> Fst 1 <> Snd 2
Snd 2
Prelude> Fst 1 <> Fst 2
Fst 2
Prelude> Snd 1 <> Fst 2
Snd 1
Prelude> Snd 1 <> Snd 2
Snd 1
```

9. ```
   newtype Combine a b =
     Combine { unCombine :: (a -> b) }
   ```

   What it should do:

   ```
   Prelude> f = Combine $ \n -> Sum (n + 1)
   Prelude> g = Combine $ \n -> Sum (n - 1)
   Prelude> unCombine (f <> g) $ 0
   Sum {getSum = 0}
   Prelude> unCombine (f <> g) $ 1
   Sum {getSum = 2}
   Prelude> unCombine (f <> f) $ 1
   Sum {getSum = 4}
   Prelude> unCombine (g <> f) $ 1
   Sum {getSum = 2}
   ```

   Hint: This function will eventually be applied to a single value of type `a`. But you'll have multiple functions that can produce a value of type `b`. How do we combine multiple values so we have a single `b`? This one will probably be tricky! Remember that the type of the value inside of `Combine` is that of a *function*. The type of functions should already have an `Arbitrary` instance that you can reuse for testing this instance.

10. ```
    newtype Comp a =
      Comp { unComp :: (a -> a) }
    ```

    Hint: We can do something that seems a little more specific and natural to functions now that the input and output types are the same.

11. Look familiar?

```
data Validation a b =
  Failure a | Success b
  deriving (Eq, Show)

instance Semigroup a =>
  Semigroup (Validation a b) where
    (<>) = undefined
```

Given this code:

```
main = do
  let failure :: String
              -> Validation String Int
      failure = Failure
      success :: Int
              -> Validation String Int
      success = Success
  print $ success 1 <> failure "blah"
  print $ failure "woot" <> failure "blah"
  print $ success 1 <> success 2
  print $ failure "woot" <> success 2
```

You should get this output:

```
Prelude> main
Success 1
Failure "wootblah"
Success 1
Success 2
```

Monoid exercises

Given a datatype, implement the `Monoid` instance. Add `Monoid` constraints to type variables where needed. For the datatypes for which you've already implemented `Semigroup` instances, you need to figure out what the identity value is.

1. Again, validate *all* of your instances with `QuickCheck`. Example scaffold is provided for the `Trivial` type:

```
data Trivial = Trivial deriving (Eq, Show)

instance Semigroup Trivial where
  (<>) = undefined

instance Monoid Trivial where
  mempty = undefined
  mappend = (<>)

type TrivAssoc =
  Trivial -> Trivial -> Trivial -> Bool

main :: IO ()
main = do
  let sa = semigroupAssoc
      mli = monoidLeftIdentity
      mlr = monoidRightIdentity
  quickCheck (sa :: TrivAssoc)
  quickCheck (mli :: Trivial -> Bool)
  quickCheck (mlr :: Trivial -> Bool)
```

2. ```
   newtype Identity a =
     Identity a deriving Show
   ```

3. ```
   data Two a b = Two a b deriving Show
   ```

4. ```
   newtype BoolConj =
     BoolConj Bool
   ```

   What it should do:

   ```
   Prelude> (BoolConj True) `mappend` mempty
   BoolConj True
   Prelude> mempty `mappend` (BoolConj False)
   BoolConj False
   ```

5. ```
   newtype BoolDisj =
     BoolDisj Bool
   ```

What it should do:

```
Prelude> (BoolDisj True) `mappend` mempty
BoolDisj True
Prelude> mempty `mappend` (BoolDisj False)
BoolDisj False
```

6. ```
   newtype Combine a b =
     Combine { unCombine :: (a -> b) }
   ```

   What it should do:

   ```
   Prelude> f = Combine $ \n -> Sum (n + 1)
   Prelude> unCombine (mappend f mempty) $ 1
   Sum {getSum = 2}
   ```

7. Hint: We can do something that seems a little more specific and natural to functions now that the input and output types are the same:

   ```
   newtype Comp a =
     Comp (a -> a)
   ```

8. This next exercise will involve doing something that will still feel a bit unnatural, and you may find it difficult. If you get it, and you haven't done much FP or Haskell before, get yourself a nice beverage. We're going to toss you the instance declaration, so you don't churn on a missing `Monoid` constraint you didn't know you need:

   ```
   newtype Mem s a =
     Mem {
       runMem :: s -> (a,s)
     }

   instance Semigroup a => Semigroup (Mem s a) where
     (<>) = undefined


   instance Monoid a => Monoid (Mem s a) where
     mempty = undefined
   ```
   ```

Given the following code:

```
f' = Mem $ \s -> ("hi", s + 1)

main = do
  let rmzero = runMem mempty 0
      rmleft = runMem (f' <> mempty) 0
      rmright = runMem (mempty <> f') 0
  print $ rmleft
  print $ rmright
  print $ (rmzero :: (String, Int))
  print $ rmleft == runMem f' 0
  print $ rmright == runMem f' 0
```

A correct `Monoid` for `Mem` should, given the above code, produce the following output:

```
Prelude> main
("hi",1)
("hi",1)
("",0)
True
True
```

Make certain your instance has output like the above, as this is sanity checking the `Monoid` identity laws for you! It's not a proof, and it's not even as good as property testing, but it'll catch the most common mistakes people make.

It's not a trick, and you don't need a `Monoid` for `s`. Yes, such a `Monoid` can and does exist. Hint: chain the `s` values from one function to the other. You'll want to check the identity laws, as a common first attempt will break them.

15.16 Definitions

1. A *monoid* is a set that is closed under an associative binary operation and has an identity element. *Closed* is the posh mathematical way of saying its type is:

```
mappend :: m -> m -> m
```

Such that your arguments and output will always inhabit the same type (set).

2. A *semigroup* is a set that is closed under an associative binary operation—and nothing else.

3. Laws are rules about how an algebra or structure should behave. These are needed in part to make abstraction over the commonalities of different instantiations of the same sort of algebra possible and *practical*. This is critical to having abstractions that aren't unpleasantly surprising.

4. An *algebra* is variously:

   a) School algebra, such as that taught in primary and secondary school. This usually entails the balancing of polynomial equations and learning how functions and graphs work.

   b) The study of number systems and operations within them. This will typically entail a particular area such as groups or rings. This is what mathematicians commonly mean by "algebra." This is sometimes disambiguated by being referred to as abstract algebra.

   c) A third and final way algebra is used is to refer to a vector space over a field with a multiplication operation.

   When Haskellers refer to algebras, they're *usually* talking about a somewhat informal notion of operations over a type and its laws, such as with semigroups, monoids, groups, semirings, and rings.

15.17 Follow-up resources

1. Simple English Wikipedia. *Algebraic structure*.
   `https://simple.wikipedia.org/wiki/Algebraic_structure`

2. Dan Piponi. *Haskell Monoids and Their Uses*.
   `http://blog.sigfpe.com/2009/01/haskell-monoids-and-their-uses.html`

Chapter 16

Functor

> Lifting is the “cheat mode” of type tetris.
>
> Michael Neale

16.1 Functor

In the last chapter, on `Monoid`, we saw what it means to talk about an algebra and turn that into a type class. This chapter, on `Functor`, and the two that follow, on `Applicative` and `Monad`, will be similar. Each of these algebras is more powerful than the last, but the general concept here will remain the same: we abstract out a common pattern, make certain it follows some laws, give it an awesome name, and wonder how we ever lived without it. `Monad` sort of steals the Haskell spotlight, but you can do more with `Functor` and `Applicative` than many people realize. Also, understanding `Functor` and `Applicative` is important to a deep understanding of `Monad`.

This chapter is all about `Functor`, and `Functor` is all about a pattern of mapping over structure. We saw `fmap` way back in the chapter on lists and noted that it worked just the same as `map`, but we *also* said back then that the difference is that you can use `fmap` with structures that *aren't lists*. Now we will begin to see what that means.

This chapter will include:

- The return of the higher-kinded types.
- `fmap`s galore and not only on lists.
- Words about type classes and constructor classes.
- Puns based on George Clinton music, probably.

16.2 What's a functor?

A functor is a way to apply a function over or around some structure that we don't want to alter. That is, we want to apply the function to the value that is "inside" some structure and leave the structure alone. That's why it is most common to introduce functor by way of mapping over lists, as we did back in Chapter 9. A function is applied to each value inside a list, and the list structure remains. A good way to relate "not altering the structure" to lists is that the length of the list after mapping a function over it will always be the same. No elements are removed or added, only transformed. The type class `Functor` generalizes this pattern so that we can use that basic idea with many types of structure, not just lists.

`Functor` is implemented in Haskell with a type class, just like `Monoid`. Other means of implementing it are possible, but this is the most convenient way to do so. The definition of the `Functor` type class looks like this:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Now let's dissect this a bit:

```
class Functor f  where
--  [1]      [2]  [3]
  fmap :: (a -> b) -> f a -> f b
-- [4]      [5]       [6]    [7]
```

1. `class` is the keyword we use, as usual, to begin the definition of a type class. `Functor` is the name of the type class we are defining.
2. Type classes in Haskell usually refer to a *type*. The letters themselves, as with type variables in type signatures, do not mean anything special. `f` is a conventional letter to choose when referring to types that have functorial structure. The `f` must be the same `f` throughout the type class definition.
3. The `where` keyword ends the declaration of the type class name and associated types. After the `where`, the operations provided by the type class are listed.
4. We begin the declaration of an operation named `fmap`.
5. The argument `a -> b` is any Haskell function of that type (remembering that it could be an `(a -> a)` function for this purpose).
6. The argument `f a` is a `Functor f` that takes a type argument `a`. That is, the `f` is a type that has an instance of the `Functor` type class.
7. The return value is `f b`. It is the *same* `f` from `f a`, while the type argument `b` *possibly but not necessarily* refers to a different type.

Before we delve into the details of how this type class works, let's see `fmap` in action, so you get a feel for what's going on first.

16.3 There's a whole lot of fmap goin' round

We have seen `fmap` before, but we haven't used it much except for with lists. With lists, it seems to do the same thing as `map`:

```
Prelude> map (\x -> x > 3) [1..6]
[False,False,False,True,True,True]
Prelude> fmap (\x -> x > 3) [1..6]
[False,False,False,True,True,True]
```

The list is, of course, one type that implements the type class `Functor`, but it seems unremarkable when it just does the same thing as `map`. However, `List` isn't the only type that implements `Functor`, and `fmap` can apply a function over or around any of those functorial structures, while `map` cannot:

```
Prelude> map (+1) (Just 1)

• Couldn't match expected type ‘[b]’
              with actual type
                ‘Maybe Integer’
• In the second argument of ‘map’,
    namely ‘(Just 1)’
  In the expression: map (+ 1) (Just 1)
  In an equation for ‘it’:
    it = map (+ 1) (Just 1)
• Relevant bindings include it :: [b]

Prelude> fmap (+1) (Just 1)
Just 2
```

Intriguing! What else?

```
-- with a tuple!
Prelude> fmap (10/) (4, 5)
(4,2.0)

-- with Either!
Prelude> rca = Right "Chris Allen"
Prelude> fmap (++ ", Esq.") rca
Right "Chris Allen, Esq."
```

We can see how the type of `fmap` specializes to different types:

```
type E e = Either e
type C e = Constant e
type I = Identity
-- Functor f =>
fmap :: (a -> b) ->    f  a ->    f  b
     :: (a -> b) ->   [ ] a ->   [ ] b
     :: (a -> b) -> Maybe a -> Maybe b
     :: (a -> b) ->   E e a ->   E e b
     :: (a -> b) ->  (e,) a ->  (e,) b
     :: (a -> b) ->     I a ->     I b
     :: (a -> b) ->   C e a ->   C e b
```

If you are using GHC 8 or newer, you can also see this for yourself in your REPL, by doing this:

```
Prelude> :set -XTypeApplications

Prelude> :type fmap @Maybe
fmap @Maybe ::
    (a -> b) -> Maybe a -> Maybe b

Prelude> :type fmap @(Either _)
fmap @(Either _) ::
    (a -> b) -> Either t a -> Either t b
```

You may have noticed in the tuple and `Either` examples that the first arguments (labeled `e` in the above chart) are ignored by `fmap`. We'll talk about why that is in just a bit. Let's first turn our attention to what makes a functor. Later, we'll come back to longer examples and expand on this considerably.

16.4 Let's talk about f, baby

As we said above, the `f` in the type class definition for `Functor` must be the same `f` throughout the entire definition, and it must refer to a type that implements the type class. This section details the practical ramifications of those facts.

The first thing we know is that `f` must have the kind `* -> *`. We talked about higher-kinded types in previous chapters, and we recall that a type constant or a fully applied type has the kind `*`. A type with kind `* -> *` is awaiting application to a type constant of kind `*`.

We know that the `f` in our `Functor` definition must have the kind `* -> *` for a couple of reasons, which we will first describe and then demonstrate:

1. Each argument (and result) in the type signature for a function must be a fully applied type. Each argument must be kind `*`.
2. The type `f` is applied to a single argument in two different places: `f a` and `f b`. Since `f a` and `f b` must each have the kind `*`, `f` by itself must be kind `* -> *`.

It's easier to see what these mean in practice by demonstrating with lots of code, so let's tear the roof off this sucker.

Shining star come into view

Every argument to the type constructor of `->` must be of kind `*`. We can verify this simply by querying the kind of the function type constructor for ourselves:

```
Prelude> :k (->)
(->) :: * -> * -> *
```

Each argument and result of every function must be a type constant, not a type constructor. Given that knowledge, we can know something about `Functor` from the type of `fmap`:

```
class Functor f where
    fmap :: (a -> b) -> f a  ->  f b
--          [  1   ]   [ 2 ]    [ 3 ]
-- 1, 2, and 3 all have the kind *
```

The type signature of `fmap` tells us that the `f` introduced by the class definition for `Functor` *must* accept a single type argument and thus be of kind `* -> *`. We can determine this even without knowing anything about the type class, which we'll demonstrate with some meaningless type classes:

```
class Sumthin a where
  s :: a -> a

class Else where
  e :: b -> f (g a b c)

class Biffy where
  slayer :: e a b
         -> (a -> c)
         -> (b -> d)
         -> e c d
```

Let's deconstruct the previous couple of examples:

```
class Sumthin a where
  s :: a -> a
--    [1]  [1]
```

1. The argument and result type are both `a`. There's nothing else, so `a` has kind `*`.

```
class Else where
  e :: b  -> f   (g  a b c)
--    [1]   [2]  [3]
```

1. This `b`, like `a` in the previous example, stands alone as the first argument to `->`, so it is kind `*`.
2. Here, `f` is the outermost type constructor for the second argument (the result type) of `->`. It takes a single argument, the type `g a b c` wrapped in parentheses. Thus, `f` has kind `* -> *`.
3. And `g` is applied to three arguments `a`, `b`, and `c`. That means it is kind `* -> * -> * -> *`, where:

```
    -- using :: to denote kind signature
    g :: * -> * -> * -> *

    -- a, b, and c are each kind *

    g :: * -> * -> *  ->    *
    g    a    b    c    (g a b c)
```

```
class Biffy where
  slayer :: e a b
--          [1]
         -> (a -> c)
--          [2]  [3]
         -> (b -> d)
         -> e c d
```

1. First, `e` is an argument to `->`, so the application of its arguments must result in kind `*`. Given that, and knowing there are two arguments, `a` and `b`, we can determine `e` is kind `* -> * -> *`.
2. This `a` is an argument to a function that takes no arguments itself, so it's kind `*`
3. The story for `c` is identical to `a`, just in another spot of the same function.

The kind checker is going to fail on the next couple of examples:

```
class Impish v where
  impossibleKind :: v -> v a

class AlsoImp v where
  nope :: v a -> v
```

Remember that the name of the variable before the `where` in a type class definition binds the occurrences of that name throughout the definition. GHC will notice that our `v` sometimes has a type argument and sometimes not, and it will call our bluff if we attempt to feed it this nonsense:

```
• Expected kind ‘k0 -> *’, but ‘v’ has
    kind ‘*’
• In the type signature:
    impossibleKind :: v -> v a
  In the class declaration for ‘Impish’

• Expecting one more argument to ‘v’
  Expected a type,
    but ‘v’ has kind ‘k0 -> *’
• In the type signature: nope :: v a -> v
  In the class declaration for ‘AlsoImp’
```

Just as GHC has type inference, it also has kind inference. And just as it does with types, it can not only infer the kinds but also validate that they're consistent and make sense.

Exercises: Be kind

Given a type signature, determine the kinds of each type variable:

1. What's the kind of `a`?

   ```
   f :: a -> a
   ```

2. What are the kinds of `b` and `T`? (The `T` is capitalized on purpose!)

   ```
   f :: a -> b a -> T (b a)
   ```

3. What's the kind of `c`?

   ```
   f :: c a b -> c b a
   ```

A shining star for you to see

So, what if our type isn't higher kinded? Let's try it with a type constant and see what happens:

```
-- functors1.hs

data FixMePls =
    FixMe
  | Pls
  deriving (Eq, Show)

instance Functor FixMePls where
  fmap =
    error
    "it doesn't matter, it won't compile"
```

Notice there are no type arguments anywhere—everything is one shining (kind) star! And if we load this file from GHCi, we'll get the following error:

```
Prelude> :l functors1.hs

• Expected kind ‘* -> *’, but ‘FixMePls’
    has kind ‘*’
• In the first argument of ‘Functor’,
    namely ‘FixMePls’
  In the instance declaration for
    ‘Functor FixMePls’
```

In fact, asking for a `Functor` for `FixMePls` doesn't really make sense. To see why this doesn't make sense, consider the types involved. Functor is:

```
fmap :: Functor f => (a -> b) -> f a -> f b
```

If we replace `f` with `FixMePls`:

```
(a -> b) -> FixMePls a -> FixMePls b
```

But `FixMePls` doesn't take type arguments, so this is really more like the following:

```
   (FixMePls -> FixMePls)
-> FixMePls
-> FixMePls
```

There's no type constructor `f` in there! The maximally polymorphic version of this is:

```
(a -> b) -> a -> b
```

So in fact, not having a type argument means this is the same as:

```
($) :: (a -> b) -> a -> b
```

In other words, without a type argument, what we have is mere function application.

Functor is function application

We just saw how trying to make a `Functor` instance for a type constant means you have function application. But, in fact, `fmap` is a specific sort of function application. Let's look at the types:

```
fmap :: Functor f => (a -> b) -> f a -> f b
```

There is also an infix operator for `fmap`. If you're using an older version of GHC, you may need to import `Data.Functor` in order to use it in the REPL. Of course, it has the same type as the prefix `fmap`:

```
-- <$> is the infix alias for fmap:
(<$>) :: Functor f
      => (a -> b)
      -> f a
      -> f b
```

Notice something?

```
(<$>) :: Functor f
      => (a -> b)  -> f a -> f b
 ($) ::  (a -> b)  ->  a  ->  b
```

`Functor` is a type class for function application "over" or "through" some structure `f` that we want to ignore and leave untouched. We'll explain "leave untouched" in more detail later, when we talk about the `Functor` laws.

A shining star for you to see what your f can truly be

Let's resume our exploration of why we need a higher-kinded `f`.

If we add a type argument to the datatype from above, we make `FixMePls` into a type constructor, and this will work:

```
-- functors2.hs

data FixMePls a =
    FixMe
  | Pls a
  deriving (Eq, Show)

instance Functor FixMePls where
  fmap =
    error
    "it doesn't matter, it won't compile"
```

Now, it'll compile!

```
Prelude> :l code/functors2.hs
[1 of 1] Compiling Main
Ok, one module loaded.
```

But wait, we don't need the error anymore! Let's fix that `Functor` instance:

```
-- functors3.hs

data FixMePls a =
    FixMe
  | Pls a
  deriving (Eq, Show)

instance Functor FixMePls where
  fmap _ FixMe = FixMe
  fmap f (Pls a) = Pls (f a)
```

Let's see how our instance lines up with the type of `fmap`:

```
fmap :: Functor f
     => (a -> b) -> f a   -> f b
fmap       f      (Pls a) = Pls (f a)
--      (a -> b)    f  a     f    b
```

While `f` is used in the type of `fmap` to represent the `Functor`, by convention, it is also used in function definitions to name an argument that is itself a function. Don't let the names fool you into thinking the `f` in our `FixMePls` instance is the same `f` as in the `Functor` type class definition.

Now, our code is happy-making!

```
Prelude> :l code/functors3.hs
[1 of 1] Compiling Main
Ok, one module loaded.
Prelude> fmap (+1) (Pls 1)
Pls 2
```

Notice that the function gets applied over and inside of the structure. This is how Haskell coders lift big heavy functions over abstract structure!

OK, let's make another mistake for the sake of being explicit. What if we change the type of our `Functor` instance from `FixMePls` to `FixMePls a`?

```
data FixMePls a =
    FixMe
  | Pls a
  deriving (Eq, Show)

instance Functor (FixMePls a) where
  fmap _ FixMe = FixMe
  fmap f (Pls a) = Pls (f a)
```

Notice that we didn't change the type—it still only takes one argument. But now that argument is part of the `f` structure. If we compile this ill-conceived code:

```
• Expected kind ‘* -> *’, but ‘FixMePls a’
    has kind ‘*’
```

```
• In the first argument of ‘Functor’,
    namely ‘(FixMePls a)’
    In the instance declaration for
      ‘Functor (FixMePls a)’
```

We end up with the same error as earlier, because applying the type constructor gives us something of kind `*` from the original kind, which is `* -> *`.

Type classes and constructor classes

You may have initially paused on the type constructor `f` in the definition of `Functor` having kind `* -> *`, but this is completely natural! In fact, earlier versions of Haskell didn't have a facility for expressing type classes in terms of higher-kinded types at all. It was developed by Mark P. Jones[1] while he was working on an implementation of Haskell called Gofer. This work generalized type classes from being usable only with types of kind `*` (also called type *constants*) to being usable with higher-kinded types, called type *constructors*, as well.

In Haskell, the two use cases have been merged, such that we don't call out constructor classes as being separate from type classes, but we think it's useful to highlight that something significant has happened here. Now we have a means of talking about the contents of types independently from the type that structures those contents. That's why we can have something like `fmap` that allows us to alter the contents of a value without altering the structure (a list, or a `Just`) around the value.

16.5 Functor laws

Instances of the `Functor` type class should abide by two basic laws. Understanding these laws is critical for understanding `Functor` and writing type class instances that are composable and easy to reason about.

Identity

The first law is the law of identity:

[1] Mark P. Jones. *A system of constructor classes: overloading and implicit higher-order polymorphism*. http://www.cs.tufts.edu/~nr/cs257/archive/mark-jones/fpca93.pdf

```
fmap id == id
```

If we `fmap` the identity function, it should have the same result as passing our value to identity. We shouldn't be changing any of the outer structure `f` that we're mapping over by mapping `id`. That's why it's the same as `id`. If we don't return a new value in the `a -> b` function mapped over the structure, then nothing should change:

```
Prelude> fmap id "Hi Julie"
"Hi Julie"
Prelude> id "Hi Julie"
"Hi Julie"
```

Try it out on a few different structures, and check for yourself.

Composition

The second law for `Functor` is the law of composition:

```
fmap (f . g)  ==  fmap f . fmap g
```

This concerns the composability of `fmap`. If we compose two functions, `f` and `g`, and `fmap` that over some structure, we should get the same result as if we mapped and then composed them:

```
Prelude> fmap ((+1) . (*2)) [1..5]
[3,5,7,9,11]
Prelude> fmap (+1) . fmap (*2) $ [1..5]
[3,5,7,9,11]
```

If an implementation of `fmap` doesn't do that, it's a broken functor.

Structure preservation

Both of these laws touch on the essential rule that functors must be structure preserving.

All we're allowed to know in the type about our instance of `Functor` implemented by `f` is that it implements `Functor`:

```
fmap :: Functor f => (a -> b) -> f a -> f b
```

The `f` is constrained by the type class `Functor`, but that is all we know about its type from this definition. As we've seen with type class-constrained polymorphism, this still allows it to be any type that has an instance of `Functor`. The core operation that this type class provides for these types is `fmap`. Because the `f` persists through the type of `fmap`, whatever the type is, we know it must be a type that can take an argument, as in `f a` and `f b`, and that it will be the "structure" we're lifting the function over when we apply it to the value inside.

16.6 The Good, the Bad, and the Ugly

We'll get a better picture of what it means for `Functor` instances to be law-abiding or law-breaking by walking through some examples. We start by defining a type constructor with one argument:

```
data WhoCares a =
    ItDoesnt
  | Matter a
  | WhatThisIsCalled
  deriving (Eq, Show)
```

This datatype only has one data constructor containing a value we could `fmap` over, and that is `Matter`. The others are nullary, so there is no value to work with inside the structure; there is only structure.

Here, we see a law-abiding instance:

```
instance Functor WhoCares where
  fmap _ ItDoesnt = ItDoesnt
  fmap _ WhatThisIsCalled =
    WhatThisIsCalled
  fmap f (Matter a) = Matter (f a)
```

Our instance must follow the identity law or else it's not a valid functor. That law dictates that `fmap id (Matter _)` must *not* touch `Matter`—that is, it must be identical to `id (Matter _)`. `Functor` is a way of lifting over structure (mapping) in such a manner that you don't have to care about the structure, because you're not *allowed* to touch the structure anyway.

Let us next consider a law-breaking instance:

```
instance Functor WhoCares where
  fmap _ ItDoesnt = WhatThisIsCalled
  fmap f WhatThisIsCalled = ItDoesnt
  fmap f (Matter a) = Matter (f a)
```

Now, we can contemplate what it means to leave the structure untouched. In this instance, we've made our structure—not the values wrapped or contained within the structure—change, by making `ItDoesnt` and `WhatThisIsCalled` do a little do-si-do. It becomes rapidly apparent why this isn't kosher at all:

```
Prelude> fmap id ItDoesnt
WhatThisIsCalled
Prelude> fmap id WhatThisIsCalled
ItDoesnt
Prelude> fmap id ItDoesnt == id ItDoesnt
False
Prelude> :{
*Main| fmap id WhatThisIsCalled ==
*Main|      id WhatThisIsCalled
*Main| :}
False
```

This certainly does not abide by the identity law. It is therefore not a valid `Functor` instance.

The law won But what if you *do* want a function that can change the value *and* the structure?

We've got wonderful news for you: that exists! It's a plain old function. Write one. Write many! The point of `Functor` is to reify and be able to talk about cases where we want to reuse functions in the presence of more structure and be transparently *oblivious* to that additional structure. We already saw that `Functor` is in some sense a special sort of function application, but since it is *special*, we want to preserve the things about it that make it different and more powerful than ordinary function application. So, we stick to the laws.

Later in this chapter, we will talk about a sort of opposite, where you can transform the structure but leave the type argument alone. This has a special name too, but there isn't a widely agreed on type class.

Composition should just work

All right, now that we've seen how we can make a `Functor` instance violate the identity law, let's take a look at how we abide by—and break!—the composition law. You may recall from above that the law looks like this:

```
fmap (f . g)  ==  fmap f . fmap g
```

Technically, this follows from `fmap id == id`, but it's worth calling out, so that we can talk about composition. This law says composing two functions lifted separately should produce the same result as if we compose the functions ahead of time and then lift the composed function all together. Maintaining this property is about preserving the composability of our code and preventing our software from doing unpleasantly surprising things. We will now consider another invalid `Functor` instance to see why this is bad news:

```
data CountingBad a =
  Heisenberg Int a
  deriving (Eq, Show)

-- super NOT OK
instance Functor CountingBad where
  fmap f     (Heisenberg n a) =
--  (a -> b)     f         a  =
    Heisenberg (n+1) (f a)
--       f             b
```

Well, what did we do here? `CountingBad` has one type argument, but `Heisenberg` has two arguments. If you look at how that lines up with the type of `fmap`, you get a hint of why this isn't going to work out well. What part of our `fmap` type does the `n` representing the `Int` argument to `Heisenberg` belong to?

We can load this horribleness up in the REPL and see that composing two `fmaps` here does not produce the same results, so the composition law doesn't hold:

```
Prelude> u = "Uncle"
Prelude> oneWhoKnocks = Heisenberg 0 u
```

```
Prelude> fmap (++" Jesse") oneWhoKnocks
Heisenberg 1 "Uncle Jesse"
Prelude> f = ((++" Jesse").(++" lol"))
Prelude> fmap f oneWhoKnocks
Heisenberg 1 "Uncle lol Jesse"
```

So far it seems fine, but what if we compose the two concatenation functions separately?

```
Prelude> j = (++ " Jesse")
Prelude> l = (++ " lol")
Prelude> fmap j . fmap l $ oneWhoKnocks
Heisenberg 2 "Uncle lol Jesse"
```

Or to make it look more like the law:

```
Prelude> f = (++" Jesse")
Prelude> g = (++" lol")
Prelude> fmap (f . g) oneWhoKnocks
Heisenberg 1 "Uncle lol Jesse"
Prelude> fmap f . fmap g $ oneWhoKnocks
Heisenberg 2 "Uncle lol Jesse"
```

We can clearly see that this:

```
fmap (f . g)  ==  fmap f . fmap g
```

Does not hold. So how do we fix it?

```
data CountingGood a =
  Heisenberg Int a
  deriving (Eq, Show)

-- Totes cool
instance Functor CountingGood where
  fmap f (Heisenberg n a) =
    Heisenberg (n) (f a)
```

Stop messing with the `Int` in `Heisenberg`. Think of anything that isn't the final type argument of our `f` in `Functor` as being part of the structure that the functions being lifted should be oblivious to.

16.7 Commonly used functors

Now that we have a sense of what `Functor` does for us and how it's meant to work, it's time to start working through some longer examples. This section is nearly all code and examples with minimal prose explanation. Interacting with these examples will help you develop an intuition for what's going on with a minimum of fuss.

We begin with a utility function:

```
Prelude> :t const
const :: a -> b -> a
Prelude> replaceWithP = const 'p'

Prelude> replaceWithP 10000
'p'
Prelude> replaceWithP "woohoo"
'p'
Prelude> replaceWithP (Just 10)
'p'
```

We'll use it with `fmap` now for various datatypes that have instances:

```
-- data Maybe a = Nothing | Just a

Prelude> fmap replaceWithP (Just 10)
Just 'p'
Prelude> fmap replaceWithP Nothing
Nothing

-- data [] a = [] | a : [a]

Prelude> fmap replaceWithP [1, 2, 3, 4, 5]
"ppppp"
Prelude> fmap replaceWithP "Ave"
"ppp"
Prelude> fmap (+1) []
[]
Prelude> fmap replaceWithP []
""
```

```
-- data (,) a b = (,) a b

Prelude> fmap replaceWithP (10, 20)
(10,'p')
Prelude> fmap replaceWithP (10, "woo")
(10,'p')
```

Again, we'll talk about why it skips the first value in the tuple in a bit. It has to do with the kindedness of tuples and the kindedness of the `f` in `Functor`.

Now, the instance for functions:

```
Prelude> negate 10
-10
Prelude> tossEmOne = fmap (+1) negate
Prelude> tossEmOne 10
-9
Prelude> tossEmOne (-10)
11
```

The functor of functions won't be discussed in great detail until we get to the chapter on `reader`, but it should look sort of familiar:

```
Prelude> tossEmOne' = (+1) . negate
Prelude> tossEmOne' 10
-9
Prelude> tossEmOne' (-10)
11
```

Now you're starting to get into the groove; let's see what else we can do with our fancy new moves.

The functors are stacked, and that's a fact

We can combine datatypes, as we've seen, usually by nesting them. We'll be using the tilde character as a shorthand for "is roughly equivalent to" throughout these examples:

```
-- lms ~ List (Maybe (String))
Prelude> n = Nothing
Prelude> w = Just "woohoo"
```

```
Prelude> ave = Just "Ave"
Prelude> lms = [ave, n, w]

Prelude> replaceWithP = const 'p'
Prelude> replaceWithP lms
'p'

Prelude> fmap replaceWithP lms
"ppp"
```

Nothing unexpected there, but we notice that `lms` has more than one `Functor` type. `Maybe` and the list type (which includes `String`) both have `Functor` instances. So, are we obligated to `fmap` only to the outermost datatype? No way, mate:

```
Prelude> (fmap . fmap) replaceWithP lms
[Just 'p',Nothing,Just 'p']

Prelude> tripFmap = fmap . fmap . fmap
Prelude> tripFmap replaceWithP lms
[Just "ppp",Nothing,Just "pppppp"]
```

Let's review in detail:

```
-- lms ~ [(Maybe String)]
Prelude> ave = Just "Ave"
Prelude> n = Nothing
Prelude> w = Just "woohoo"
Prelude> lms = [ave, n, w]

Prelude> replaceWithP lms
'p'

Prelude> :t replaceWithP lms
replaceWithP lms :: Char

-- In:
replaceWithP lms

-- replaceWithP's input type is:
```

```
[(Maybe String)]

-- The output type is Char

-- So applying
replaceWithP

-- to
lms

-- accomplishes
[(Maybe String) -> Char]
```

The output type of `replaceWithP` is always the same.
If we do this:

```
Prelude> fmap replaceWithP lms
"ppp"
```

`fmap` is going to leave the list structure intact around our result:

```
Prelude> :t fmap replaceWithP lms
fmap replaceWithP lms :: [Char]
```

Here's the X-ray view:

```
-- In:
fmap replaceWithP lms

-- replaceWithP's input type is:
Maybe String

-- The output type is Char

-- So applying
fmap replaceWithP

-- to
lms
```

```
-- accomplishes:
[(Maybe String)] -> [Char]

[Char] ~ String
```

What if we lift twice?

Keep on stacking them up:

```
Prelude> (fmap . fmap) replaceWithP lms
[Just 'p',Nothing,Just 'p']

Prelude> :t (fmap . fmap) replaceWithP lms
(fmap . fmap) replaceWithP lms
  :: [Maybe Char]
```

And the X-ray view:

```
-- In:
(fmap . fmap) replaceWithP lms

-- replaceWithP's input type is:
-- String aka [Char]

-- The output type is Char

-- So applying
(fmap . fmap) replaceWithP

-- to
lms

-- accomplishes
[(Maybe String)] -> [(Maybe Char)]
```

Wait, how does that even type check? It may not seem obvious at first how `(fmap . fmap)` could type check. We're going to ask you to work through the types. You might prefer to write it out with pen and paper, as Julie does, or type it all out in a text editor, as Chris does. We'll help you out by providing the type signatures. Since the two

`fmap` functions being composed could have different types, we'll make the type variables for each function unique. Start by substituting the type of each `fmap` for each of the function types in the `(.)` signature:

```
(.)  :: (b -> c) -> (a -> b) -> a -> c
--        fmap        fmap
fmap :: Functor f => (m -> n) -> f m -> f n
fmap :: Functor g => (x -> y) -> g x -> g y
```

It might also be helpful to query the type of `(fmap . fmap)` to get an idea of what your end type should look like (modulo different type variables).

Lift me baby one more time

We have another layer we can lift over if we wish:

```
Prelude> tripFmap = fmap . fmap . fmap
Prelude> tripFmap replaceWithP lms
[Just "ppp",Nothing,Just "pppppp"]

Prelude> :t tripFmap replaceWithP lms
(fmap . fmap . fmap) replaceWithP lms
  :: [Maybe [Char]]
```

And the X-ray view:

```
-- In
(fmap . fmap . fmap) replaceWithP lms

-- replaceWithP's input type is:
-- Char
-- because we lifted over
-- the [] of [Char]

-- The output type is Char

-- So applying
(fmap . fmap . fmap) replaceWithP
```

```
-- to
lms

-- accomplishes
List (Maybe String) -> List (Maybe String)
```

So, we see there's a pattern.

The real type of thing going down

We saw the pattern above, but for clarity, we'll summarize here before moving on:

```
Prelude> fmap replaceWithP lms
"ppp"

Prelude> (fmap . fmap) replaceWithP lms
[Just 'p',Nothing,Just 'p']

Prelude> tripFmap = fmap . fmap . fmap
Prelude> tripFmap replaceWithP lms
[Just "ppp",Nothing,Just "pppppp"]
```

Let's summarize the *types*, too, to validate our understanding:

```
-- replacing the type synonym String
-- with the underlying type [Char]
-- intentionally

replaceWithP' :: [Maybe [Char]] -> Char
replaceWithP' = replaceWithP

[Maybe [Char]] -> [Char]
[Maybe [Char]] -> [Maybe Char]
[Maybe [Char]] -> [Maybe [Char]]
```

Pause for a second, and make sure you've understood everything we've done so far. If not, play with the examples until it feels comfortable.

Get on up and get down

We'll work through the same idea but with more funky structure to lift over:

```
-- lmls ~ List (Maybe (List String))

Prelude> ha = Just ["Ha", "Ha"]
Prelude> lmls = [ha, Nothing, Just []]

Prelude> (fmap . fmap) replaceWithP lmls
[Just 'p',Nothing,Just 'p']

Prelude> tripFmap = fmap . fmap . fmap
Prelude> tripFmap replaceWithP lmls
[Just "pp",Nothing,Just ""]

Prelude> (tripFmap.fmap) replaceWithP lmls
[Just ["pp","pp"],Nothing,Just []]
```

See if you can trace the changing result types, as we did above.

One more round for the P-Funkshun

For those who like their funk uncut, here's another look at the changing types that result from lifting over multiple layers of functorial structure, with a slightly higher resolution. We start this time from a source file:

```
module ReplaceExperiment where

replaceWithP :: b -> Char
replaceWithP = const 'p'

lms :: [Maybe [Char]]
lms = [Just "Ave", Nothing, Just "woohoo"]
```

We can make the argument more specific:

```
replaceWithP' :: [Maybe [Char]] -> Char
replaceWithP' = replaceWithP
```

What happens if we lift it?

```
Prelude> :t fmap replaceWithP
fmap replaceWithP
  :: Functor f => f a -> f Char
```

We can use this type:

```
liftedReplace :: Functor f => f a -> f Char
liftedReplace = fmap replaceWithP
```

But we can assert an even more specific type for `liftedReplace`!

```
liftedReplace' :: [Maybe [Char]] -> [Char]
liftedReplace' = liftedReplace
```

The `[]` around Char is the `f` of `f Char`, or the structure we lift over. The `f` of `f a` is the outermost `[]` in [Maybe [Char]]. So, `f` is instantiated to `[]` when we make the type more specific, whether by applying it to a value of type `[Maybe [Char]]` or by means of explicitly writing `liftedReplace'`.

Stay on the scene like an fmap machine

What if we lift it twice?

```
Prelude> :t (fmap . fmap) replaceWithP
(fmap . fmap) replaceWithP
  :: (Functor f1, Functor f)
    => f (f1 a) -> f (f1 Char)
```

Using that:

```
twiceLifted :: (Functor f1, Functor f) =>
               f (f1 a) -> f (f1 Char)
twiceLifted = (fmap . fmap) replaceWithP
```

Making it more specific:

```
twiceLifted' :: [Maybe [Char]]
             -> [Maybe Char]
twiceLifted' = twiceLifted
-- f  ~ []
-- f1 ~ Maybe
```

Thrice?

```
Prelude> rWP = replaceWithP
Prelude> :t (fmap . fmap . fmap) rWP
(fmap . fmap . fmap) replaceWithP
  :: (Functor f2, Functor f1, Functor f)
  => f (f1 (f2 a)) -> f (f1 (f2 Char))

thriceLifted ::
     (Functor f2, Functor f1, Functor f)
  => f (f1 (f2 a)) -> f (f1 (f2 Char))
thriceLifted =
  (fmap . fmap . fmap) replaceWithP
```

More specific or "concrete":

```
thriceLifted' :: [Maybe [Char]]
              -> [Maybe [Char]]
thriceLifted' = thriceLifted
-- f  ~ []
-- f1 ~ Maybe
-- f2 ~ []
```

Now, we can print the results from our expressions and compare them:

```
main :: IO ()
main = do
  putStr "replaceWithP' lms:   "
  print (replaceWithP' lms)

  putStr "liftedReplace lms:   "
  print (liftedReplace lms)
```

```
  putStr "liftedReplace' lms:  "
  print (liftedReplace' lms)

  putStr "twiceLifted lms:     "
  print (twiceLifted lms)

  putStr "twiceLifted' lms:    "
  print (twiceLifted' lms)

  putStr "thriceLifted lms:    "
  print (thriceLifted lms)

  putStr "thriceLifted' lms:   "
  print (thriceLifted' lms)
```

Be sure to type all this into a file, load it in GHCi, and run `main` to see what output results. Then, modify the types and the code-based ideas and guesses of what should and shouldn't work. Forming hypotheses, creating experiments based on them or modifying existing experiments, and validating them is a *critical* part of becoming comfortable with abstractions like `Functor`!

Exercises: Heavy lifting

Add `fmap`, parentheses, and function composition to each expression as needed for the expression to type check and produce the expected result. It may not always need to go in the same place, so don't become complacent:

1. ```
   a = (+1) $ read "[1]" :: [Int]
   ```

   Expected result:

   ```
   Prelude> a
   [2]
   ```

2. ```
   b = (++ "lol") (Just ["Hi,", "Hello"])
   ```

   ```
   Prelude> b
   Just ["Hi,lol","Hellolol"]
   ```

3. ```
   c = (*2) (\x -> x - 2)
   ```

   ```
   Prelude> c 1
   -2
   ```

4. ```
   d =
     ((return '1' ++) . show)
     (\x -> [x, 1..3])
   ```

   ```
   Prelude> d 0
   "1[0,1,2,3]"
   ```

5. ```
   e :: IO Integer
   e = let ioi = readIO "1" :: IO Integer
           changed = read ("123"++) show ioi
       in (*3) changed
   ```

   ```
   Prelude> e
   3693
   ```

16.8 Transforming the unapplied type argument

We've seen that `f` must be a higher-kinded type and that `Functor` instances must abide by two laws, and we've played around with some basic `fmap`ping. We know that the goal of `fmap`ping is to leave the outer structure untouched while transforming the type arguments inside.

Way back in the beginning, we noticed that when we `fmap` over a tuple, it only transforms the second argument (the `b`). We saw a similar thing when we mapped over an `Either` value, and we said we'd come back to this topic. Then, we saw another hint of it above in the `Heisenberg` example. Now, the time has come to talk about what happens to the other type arguments (if any) when we can only transform the innermost.

We'll start with a couple of canonical types:
   ```

```
data Two a b =
  Two a b
  deriving (Eq, Show)

data Or a b =
    First a
  | Second b
  deriving (Eq, Show)
```

You may recognize these as `(,)` and `Either` recapitulated, the generic product and sum types, from which any combination of *and* and *or* may be made. But these are both kind `* -> * -> *`, which isn't compatible with `Functor`, so how do we write `Functor` instances for them?

These wouldn't work, because `Two` and `Or` have the wrong kind:

```
instance Functor Two where
  fmap = undefined

instance Functor Or where
  fmap = undefined
```

We know that we can partially apply functions, and we've seen previously that we can do this:

```
Prelude> :k Either
Either :: * -> * -> *
Prelude> :k Either Integer
Either Integer :: * -> *
Prelude> :k Either Integer String
Either Integer String :: *
```

That has the effect of applying out some of the arguments, reducing the kindedness of the type. Previously, we demonstrated this by applying the type constructor to concrete types; however, you can also apply it to a type variable that represents a type constant to produce the same effect.

So, to fix the kind incompatibility for our `Two` and `Or` types, we apply one of the arguments of each type constructor, giving us kind

`* -> *`. We use `a` for clarity, so you can see more readily which type is applied out, but the letter doesn't matter:

```
instance Functor (Two a) where
  fmap = undefined

instance Functor (Or a) where
  fmap = undefined
```

These will pass the type checker already, but we still need to write the implementations of `fmap` for both, so let's proceed. First, we'll turn our attention to `Two`:

```
instance Functor (Two a) where
  fmap f (Two a b) = Two $ (f a) (f b)
```

This won't fly, because the `a` is part of the functorial structure (the `f`). We're not supposed to touch anything in the `f` referenced in the type of `fmap`, so we can't apply the function (named `f` in our `fmap` definition) to the `a`, because the `a` is now untouchable:

```
fmap :: Functor f => (a -> b) -> f a -> f b
```

Here, `f` is `(Two a)`, because:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b

instance Functor (Two a) where

-- We can make the type of fmap
-- more concrete, like this:
:: (a -> b) -> (Two z) a -> (Two z) b
```

So, to fix our `Functor` instance, we have to leave the left value (it's part of the structure of `f`) in `Two` alone and have our function only apply to the innermost value, in this case named `b`:

```
instance Functor (Two a) where
  fmap f (Two a b) = Two a (f b)
```

Then with `Or`, we're dealing with the independent possibility of two different values and types, but the same basic constraint applies:

```
instance Functor (Or a) where
  fmap _ (First a)  = First a
  fmap f (Second b) = Second (f b)
```

We've applied out the first argument, so now it's part of the `f`. The function we're mapping around that structure can only transform the innermost argument.

16.9 QuickChecking Functor instances

We know the `Functor` laws are the following:

```
fmap id      = id
fmap (p . q) = (fmap p) . (fmap q)
```

We can turn those into the following `QuickCheck` properties:

```
functorIdentity :: (Functor f, Eq (f a)) =>
                       f a
                    -> Bool
functorIdentity f =
  fmap id f == f

functorCompose :: (Eq (f c), Functor f) =>
                      (a -> b)
                   -> (b -> c)
                   -> f a
                   -> Bool
functorCompose f g x =
  (fmap g (fmap f x)) == (fmap (g . f) x)
```

As long as we provide concrete instances, we can now run these to test them:

```
Prelude> :{
*Main| let f :: [Int] -> Bool
*Main|     f x = functorIdentity x
```

```
*Main| :}
Prelude> quickCheck f
+++ OK, passed 100 tests.

Prelude> c = functorCompose (+1) (*2)
Prelude> li x = c (x :: [Int])

Prelude> quickCheck li
+++ OK, passed 100 tests.
```

Groovy.

16.10 Exercises: Instances of Func

Implement `Functor` instances for the following datatypes. Use the `QuickCheck` properties we showed you to validate them:

1. `newtype Identity a = Identity a`
2. `data Pair a = Pair a a`
3. `data Two a b = Two a b`
4. `data Three a b c = Three a b c`
5. `data Three' a b = Three' a b b`
6. `data Four a b c d = Four a b c d`
7. `data Four' a b = Four' a a a b`
8. Can you implement one for this type? Why? Why not?

   ```
   data Trivial = Trivial
   ```

Doing these exercises is *critical* to understanding how `Functor` works—do not skip past them!

16.11 Ignoring possibilities

We've already touched on the `Maybe` and `Either` functors. Now we'll examine in a bit more detail what those do for us. As the title of this section suggests, the `Functor` instances for these datatypes are handy for times you intend to ignore the left cases, which are typically your error or failure cases. Because `fmap` doesn't touch those cases, you can map your function right to the values that you intend to work with and ignore those failure cases.

Maybe

Let's start with some ordinary pattern matching on `Maybe`:

```
incIfJust :: Num a => Maybe a -> Maybe a
incIfJust (Just n) = Just $ n + 1
incIfJust Nothing = Nothing

showIfJust :: Show a
           => Maybe a
           -> Maybe String
showIfJust (Just s) = Just $ show s
showIfJust Nothing = Nothing
```

Well, that's boring, and there's some redundant structure. For one thing, they have the `Nothing` case in common:

```
someFunc Nothing = Nothing
```

Then, they're applying some function to the value if it's a `Just`:

```
someFunc (Just x) = Just $ someOtherFunc x
```

What happens if we use `fmap`?

```
incMaybe :: Num a => Maybe a -> Maybe a
incMaybe m = fmap (+1) m

showMaybe :: Show a
          => Maybe a
          -> Maybe String
showMaybe s = fmap show s
```

That appears to have cleaned things up a bit. Does it still work?

```
Prelude> incMaybe (Just 1)
Just 2
Prelude> incMaybe Nothing
Nothing
Prelude> showMaybe (Just 9001)
Just "9001"
Prelude> showMaybe Nothing
Nothing
```

Yeah, `fmap` has no reason to concern itself with the `Nothing`—there's no value there for it to operate on, so this all seems to be working properly.

But we can abstract this a bit more. For one thing, we can *eta reduce* these functions. That is, we can rewrite them without naming the arguments:

```
incMaybe'' :: Num a => Maybe a -> Maybe a
incMaybe'' = fmap (+1)

showMaybe'' :: Show a
            => Maybe a
            -> Maybe String
showMaybe'' = fmap show
```

And they don't even really have to be specific to `Maybe`! `fmap` works for all datatypes with a `Functor` instance! We can query the type of the expressions in GHCi and see for ourselves the more generic types:

```
Prelude> :t fmap (+1)
fmap (+1)
  :: (Functor f, Num b) => f b -> f b

Prelude> :t fmap show
fmap show
  :: (Functor f, Show a) => f a -> f String
```

With that, we can rewrite them as much more generic functions:

```
-- "lifted" because they're lifted
-- over some structure f

liftedInc :: (Functor f, Num b)
          => f b -> f b
liftedInc = fmap (+1)

liftedShow :: (Functor f, Show a)
           => f a -> f String
liftedShow = fmap show
```

And they have the same behavior, as always:

```
Prelude> liftedInc (Just 1)
Just 2
Prelude> liftedInc Nothing
Nothing

Prelude> liftedShow (Just 1)
Just "1"
Prelude> liftedShow Nothing
Nothing
```

Making them more polymorphic in the type of the functorial structure means they're more reusable now:

```
Prelude> liftedInc [1..5]
[2,3,4,5,6]

Prelude> liftedShow [1..5]
["1","2","3","4","5"]
```

Exercise: Possibly

Write a `Functor` instance for a datatype identical to `Maybe`. We'll use our own datatype, because `Maybe` already has a `Functor` instance and we cannot make a duplicate one:

```
data Possibly a =
    LolNope
  | Yeppers a
  deriving (Eq, Show)

instance Functor Possibly where
  fmap = undefined
```

If it helps, you're basically writing the following function:

```
applyIfJust :: (a -> b)
            -> Maybe a
            -> Maybe b
```

Either

The `Maybe` type solves some problems for Haskellers, but it doesn't solve all of them. As we saw previously, in Chapter 12, sometimes we want to preserve the reason *why* a computation fails rather than only the information *that* it fails. And for that, we use `Either`.

By this point, you know that `Either` has a `Functor` instance in `base` for grateful programmers to use. So let's put it to use. We'll stick to the same pattern we used for demonstrating `Maybe`, for the sake of clarity:

```
incIfRight :: Num a
           => Either e a
           -> Either e a
incIfRight (Right n) = Right $ n + 1
incIfRight (Left  e) = Left e

showIfRight :: Show a
            => Either e a
            -> Either e String
showIfRight (Right s) = Right $ show s
showIfRight (Left  e) = Left e
```

Once again, we can simplify these using `fmap`, so we don't have to address the case of leaving the error value alone:

```
incEither :: Num a
          => Either e a
          -> Either e a
incEither m = fmap (+1) m

showEither :: Show a
           => Either e a
           -> Either e String
showEither s = fmap show s
```

And again, we can eta-contract to drop the obvious argument:

```
incEither' :: Num a
           => Either e a
           -> Either e a
incEither' = fmap (+1)

showEither' :: Show a
            => Either e a
            -> Either e String
showEither' = fmap show
```

And once *again*, we are confronted with functions that really didn't need to be specific to `Either` at all:

```
-- f ~ Either e

liftedInc :: (Functor f, Num b)
          => f b -> f b
liftedInc = fmap (+1)

liftedShow :: (Functor f, Show a)
           => f a -> f String
liftedShow = fmap show
```

Take a few moments to play around with this code, and note how it works.

Short exercise

1. Write a `Functor` instance for a datatype identical to `Either`. We'll use our own datatype, because `Either` has a `Functor` instance:

   ```
   data Sum a b =
       First a
     | Second b
     deriving (Eq, Show)

   instance Functor (Sum a) where
     fmap = undefined
   ```

   Your hint for this one is that you're actually writing the following function:

   ```
   applyIfSecond :: (a -> b)
                 -> (Sum e) a
                 -> (Sum e) b
   ```

2. Why is a `Functor` instance that applies a function only to `First`, `Either`'s `Left`, impossible? We covered this earlier.

16.12 A somewhat surprising functor

There's a datatype named `Const` or `Constant`—you'll see both names, depending on which library you use. `Constant` has a valid `Functor`, but the behavior of the `Functor` instance may surprise you a bit. First, let's look at the `const` *function*, and then we'll look at the datatype:

```
Prelude> :t const
const :: a -> b -> a
Prelude> a = const 1
Prelude> a 1
1
Prelude> a 2
1
Prelude> a 3
1
Prelude> a "blah"
```

```
1
Prelude> a id
1
```

With this concept in mind, let's have a look at the `Constant` datatype, which is similar. `Constant` looks like this:

```
newtype Constant a b =
  Constant { getConstant :: a }
  deriving (Eq, Show)
```

One thing we notice about this type is that the type parameter `b` is a *phantom* type. It has no corresponding `witness` at the value/term level. This is a concept and tactic we'll explore more later, but for now, we can see how it echoes the function `const`:

```
Prelude> Constant 2
Constant {getConstant = 2}
```

Despite `b` being a phantom type, `Constant` is kind `* -> * -> *`, and that is not a valid `Functor`. So how do we get one? Well, there's only one thing we can do with a type constructor, just as with functions: apply it. So we *do* have a `Functor` for `Constant a`, but it's not `Constant` alone. It has to be `Constant a` and not `Constant a b`, because `Constant a b` would be kind `*`.

Let's look at the implementation of `Functor` for `Constant`:

```
instance Functor (Constant m) where
    fmap _ (Constant v) = Constant v
```

Looks like identity right? Let's use this in the REPL and run it through the `Functor` laws:

```
Prelude> const 2 (getConstant (Constant 3))
2
Prelude> fmap (const 2) (Constant 3)
Constant {getConstant = 3}

Prelude> gc = getConstant
Prelude> c = Constant 3
```

```
Prelude> gc $ fmap (const 2) c
3
Prelude> gc $ fmap (const "blah") c
3
```

When you `fmap` the `const` function over the `Constant` type, the first argument to `const` is never used, because the partially applied `const` is itself never used. The first type argument to `Constant`'s type constructor is in the part of the structure that `Functor` skips over. The second argument to the `Constant` type constructor is the phantom type variable `b`, which has no value or term-level witness in the datatype. Since there are no values of the type the `Functor` is supposed to be mapping, we have nothing we're allowed to apply the function to, so we never use the `const` expressions.

But does this adhere to the `Functor` laws?

```
-- Testing identity
Prelude> getConstant (id (Constant 3))
3
Prelude> getConstant (fmap id (Constant 3))
3

-- Composition of the const function
Prelude> ((const 3) . (const 5)) 10
3
Prelude> ((const 5) . (const 3)) 10
5

-- Composition
Prelude> fc = fmap (const 3)
Prelude> fc' = fmap (const 5)
Prelude> separate = fc . fc'
Prelude> c = const 3
Prelude> c' = const 5
Prelude> fused = fmap (c . c')
Prelude> cw = Constant "WOOHOO"
Prelude> getConstant $ separate $ cw
"WOOHOO"
Prelude> cdr = Constant "Dogs rule"
```

```
Prelude> getConstant $ fused $ cdr
"Dogs rule"
```

`(Constant a)` is `* -> *`, which you need for the `Functor`, but now you're mapping over that `b` and not the `a`.

This is a mere cursory check, not a proof that this is a valid `Functor`. Most assurances of correctness that programmers use exist on a gradient and aren't proper proofs. Despite seeming a bit pointless, however, `Constant` is a lawful `Functor`.

16.13 More structure, more functors

At times, the structure of our types may require that we also have a `Functor` instance for an intermediate type layer. We'll demonstrate this using the following datatype:

```
data Wrap f a =
  Wrap (f a)
  deriving (Eq, Show)
```

Notice that our `a` here is an argument to the `f`. So how are we going to write a `Functor` instance for this?

```
instance Functor (Wrap f) where
  fmap f (Wrap fa) = Wrap (f fa)
```

This won't work, because there's this `f` that we're not hopping over, and `a` (the value `fmap` should be applying the function to) is an argument to `f`—the function can't apply to the `f` that is wrapping `a`.

```
instance Functor (Wrap f) where
  fmap f (Wrap fa) = Wrap (fmap f fa)
```

Here, we don't know what type `f` is, and it could be anything, but it needs to be a type that has a `Functor` instance so that we can `fmap` over it. So we add a constraint:

```
instance Functor f
      => Functor (Wrap f) where
  fmap f (Wrap fa) = Wrap (fmap f fa)
```

And if we load up the final instance, we can use this wrapper type:

```
Prelude> fmap (+1) (Wrap (Just 1))
Wrap (Just 2)

Prelude> fmap (+1) (Wrap [1, 2, 3])
Wrap [2,3,4]
```

It should work for any `Functor`. If we pass it something that isn't?

```
Prelude> n = 1 :: Integer
Prelude> fmap (+1) (Wrap n)

• Couldn't match expected type ‘f b’ with
    actual type ‘Integer’
• In the first argument of ‘Wrap’,
    namely ‘n’
  In the second argument of ‘fmap’,
    namely ‘(Wrap n)’
  In the expression: fmap (+ 1) (Wrap n)
• Relevant bindings include
    it :: Wrap f b
```

The number by itself doesn't offer the additional structure that `Wrap` needs to work as a `Functor`. It's expecting to be able to `fmap` over some `f` independent of an `a`, and this isn't the case with any type constant such as `Integer`.

16.14 IO Functor

We saw the `IO` type in previous chapters already, but we didn't do much with it save to print text or ask for string input from the user. The `IO` type will get a full chapter of its own later in the book. It is an abstract datatype; there are no data constructors that you're permitted to pattern match on, so the type classes `IO` provides are the only way you can work with values of type `IO a`. One of the simplest provided is `Functor`:

```
-- getLine :: IO String
-- read :: Read a => String -> a

getInt :: IO Int
getInt = fmap read getLine
```

`Int` has a `Read` instance, and `fmap` lifts `read` over the `IO` type. A way you can read `getLine` here is that it's not a `String` but rather *a way to obtain a string*. `IO` doesn't guarantee that effects will be performed, but it does mean that they *could* be performed. Here, the side effect is needing to block and wait for user input via the standard input stream the OS provides:

```
Prelude> getInt
10
10
```

We type 10 and hit enter. GHCi prints `IO` values unless the type is `IO ()`, in which case it hides the unit value, because it's meaningless:

```
Prelude> fmap (const ()) getInt
10
```

The "10" in the GHCi session above is from typing 10 and hitting enter. GHCi isn't printing any result after that, because we're replacing the `Int` value we read from a `String`. That information is getting dropped on the floor, because we apply `const ()` to the contents of the `IO Int`. If we ignore the presence of `IO`, it's as if we do this:

```
Prelude> getInt = 10 :: Int
Prelude> const () getInt
()
```

GHCi, as a matter of convenience and design, will not print any value of type `IO ()` on the assumption that the IO action you evaluate is evaluated for effects and because the unit value cannot communicate anything. We can use the `return` function (seen earlier, explained later) to lift a unit value in `IO` and reproduce this behavior:

```
Prelude> return 1 :: IO Int
1
Prelude> ()
()
Prelude> return () :: IO ()
Prelude>
```

What if we want to do something more useful? We can `fmap` any function we want over `IO`:

```
Prelude> fmap (+1) getInt
10
11

Prelude> fmap (++ " and me too!") getLine
hello
"hello and me too!"
```

We can also use `do` syntax to do the same thing that we're doing with `Functor` here:

```
meTooIsm :: IO String
meTooIsm = do
  input <- getLine
  return (input ++ "and me too!")

bumpIt :: IO Int
bumpIt = do
  intVal <- getInt
  return (intVal + 1)
```

But if `fmap f` suffices for what you're doing, that's usually shorter and clearer. It's perfectly all right and quite common to start with a more verbose form of some expression and then clean it up after you've got something that works.

16.15 What if we want to do something different?

We've talked about `Functor` as a means of lifting functions over structure so that we only transform the contents, leaving the structure

alone. What if we only want to transform the *structure*, instead, and leave the *type argument* to that structure or type constructor alone? With this, we've arrived at *natural transformations*. We can attempt to put together a type to express what we want:

```
nat :: (f -> g) -> f a -> g a
```

This type is impossible, because we can't have higher-kinded types as argument types to the function type. What's the problem, though? It looks like the type signature for `fmap`, doesn't it? Yet `f` and `g` in `f -> g` are higher-kinded types. They must be, because they are the same `f` and `g` that, later in the type signature, are taking arguments. But in those places they are applied to their arguments and so have kind `*`.

So, we make a modest change to fix it:

```
{-# LANGUAGE RankNTypes #-}

type Nat f g = forall a . f a -> g a
```

In a sense, we're doing the opposite of what a `Functor` does. We're transforming the structure, preserving the values as they are. We won't explain it fully here, but the quantification of `a` in the right-hand side of the declaration allows us to obligate all functions of this type to be oblivious to the contents of the structures `f` and `g` in much the same way that the identity function cannot do anything but return the argument it is given.

Syntactically, it lets us avoid talking about `a` in the type of `Nat`—which is what we want, as we shouldn't *have* any specific information about the contents of `f` and `g`, because we're only supposed to be performing a structural transformation, not a fold.

If you try to elide the `a` from the type arguments without quantifying it separately, you'll get an error:

```
Prelude> type Nat f g = f a -> g a

Not in scope: type variable ‘a’
```

We can add the quantifier, but if we forget to turn on `RankNTypes` (or `Rank2Types`), it won't work:

```
Prelude> :{
*Main| type Nat f g =
*Main|        forall a . f a -> g a
*Main| :}
Illegal symbol '.' in type
Perhaps you intended to use RankNTypes or a
similar language extension to enable
explicit-forall syntax:
  forall <tvs>. <type>
```

If we turn on `RankNTypes`, it works fine:

```
Prelude> :set -XRankNTypes
Prelude> :{
*Main| type Nat f g =
*Main|        forall a . f a -> g a
*Main| :}
Prelude>
```

To see an example of what the quantification prevents, consider the following:

```
type Nat f g = forall a . f a -> g a

-- This'll work
maybeToList :: Nat Maybe []
maybeToList Nothing = []
maybeToList (Just a) = [a]

-- This will not work, not allowed
degenerateMtl :: Nat Maybe []
degenerateMtl Nothing = []
degenerateMtl (Just a) = [a + 1]
```

What if we use a version of `Nat` that mentions `a` in the type?

```
module BadNat where


type Nat f g a = f a -> g a
```

```
-- This'll work
maybeToList :: Nat Maybe [] a
maybeToList Nothing = []
maybeToList (Just a) = [a]
```

This will work if we tell it that `a` is `Num a => a`:

```
degenerateMtl :: Num a => Nat Maybe [] a
degenerateMtl Nothing = []
degenerateMtl (Just a) = [a + 1]
```

That last example should *not* work and is not a good way to think about natural transformation. Part of software development is being precise, and when we talk about natural transformations, we're saying as much about what we *don't* want as we are about what we *do* want. In this case, the invariant we want to preserve is that the function cannot do anything mischievous with the values. If you want to transform the values, write a plain old fold!

We're going to return to the topic of natural transformations in the next chapter, so cool your jets for now.

16.16 Functors are unique to a datatype

In Haskell, `Functor` instances will be unique for a given datatype. We saw that this isn't true for `Monoid`—we use newtypes to preserve the unique pairing of a `Monoid` instance to a type. But `Functor` instances will be unique for a datatype, in part because of parametricity, in part because arguments to type constructors are applied in order of definition. In a hypothetical not-Haskell language, the following might be possible:

```
data Tuple a b =
  Tuple a b
  deriving (Eq, Show)

-- this is impossible in Haskell
instance Functor (Tuple ? b) where
  fmap f (Tuple a b) = Tuple (f a) b
```

There are essentially two ways to address this. One is to flip the arguments to the type constructor. The other is to make a new datatype using a `Flip` newtype:

```
{-# LANGUAGE FlexibleInstances #-}

module FlipFunctor where

data Tuple a b =
  Tuple a b
  deriving (Eq, Show)

newtype Flip f a b =
  Flip (f b a)
  deriving (Eq, Show)

-- this works, goofy as it looks
instance Functor (Flip Tuple a) where
  fmap f (Flip (Tuple a b)) =
    Flip $ Tuple (f a) b
```

```
Prelude> fmap (+1) (Flip (Tuple 1 "blah"))
Flip (Tuple 2 "blah")
```

However, `Flip Tuple a b` is a distinct type from `Tuple a b`, even if it's only there to provide for different `Functor` instance behavior.

16.17 Chapter exercises

Determine whether a valid `Functor` can be written for the datatype provided:

1. ```
   data Bool =
       False | True
   ```

2. ```
   data BoolAndSomethingElse a =
       False' a | True' a
   ```

3. ```
   data BoolAndMaybeSomethingElse a =
       Falsish | Truish a
   ```
   ```

4. Use the kinds to guide you on this one—don't get too hung up on the details:

```
newtype Mu f = InF { outF :: f (Mu f) }
```

5. Again, follow the kinds, and ignore the unfamiliar parts:

```
import GHC.Arr

data D =
  D (Array Word Word) Int Int
```

Rearrange the arguments to the type constructor of the datatype so the `Functor` instance works:

1. ```
   data Sum a b =
     First a
   | Second b

   instance Functor (Sum e) where
     fmap f (First a)  = First (f a)
     fmap f (Second b) = Second b
   ```

2. ```
   data Company a b c =
     DeepBlue a c
   | Something b

   instance Functor (Company e e') where
     fmap f (Something b) = Something (f b)
     fmap _ (DeepBlue a c) = DeepBlue a c
   ```

3. ```
   data More a b =
     L a b a
   | R b a b
   deriving (Eq, Show)

   instance Functor (More x) where
     fmap f (L a b a') = L (f a) b (f a')
     fmap f (R b a b') = R b (f a) b'
   ```
   ```

Keeping in mind that it should result in a `Functor` that does the following:

```
Prelude> fmap (+1) (L 1 2 3)
L 2 2 4
Prelude> fmap (+1) (R 1 2 3)
R 1 3 3
```

Write `Functor` instances for the following datatypes:

1. ```
   data Quant a b =
     Finance
   | Desk a
   | Bloor b
   ```

2. No, it's not interesting by itself:

   ```
   data K a b =
     K a
   ```

3. ```
   {-# LANGUAGE FlexibleInstances #-}

   newtype Flip f a b =
     Flip (f b a)
     deriving (Eq, Show)

   newtype K a b =
     K a

   -- This should remind you of an
   -- instance you've written before
   instance Functor (Flip K a) where
     fmap = undefined
   ```

4. ```
   data EvilGoateeConst a b =
     GoatyConst b

   -- You thought you'd escaped the goats
   -- by now didn't you? Nope.
   ```
   ```

No, it doesn't do anything interesting. No magic here or in the previous exercise. If it works, you succeeded.

5. Do you need something extra to make the instance work?

```
data LiftItOut f a =
  LiftItOut (f a)
```

6. ```
   data Parappa f g a =
     DaWrappa (f a) (g a)
   ```

7. Don't ask for more type class instances than you need. You can let GHC tell you what to do:

```
data IgnoreOne f g a b =
  IgnoringSomething (f a) (g b)
```

8. ```
   data Notorious g o a t =
     Notorious (g o) (g a) (g t)
   ```

9. You'll need to use recursion:

```
data List a =
    Nil
  | Cons a (List a)
```

10. A tree of goats forms the Goat-Lord, a fearsome poly-creature:

```
data GoatLord a =
    NoGoat
  | OneGoat a
  | MoreGoats (GoatLord a)
              (GoatLord a)
              (GoatLord a)
-- A VERITABLE HYDRA OF GOATS
```

11. You'll use an extra functor for this one, although your solution might do it monomorphically without using `fmap`. Keep in mind that you will probably not be able to validate this one in the usual manner. Do your best to make it work:[2]

```
data TalkToMe a =
    Halt
  | Print String a
  | Read (String -> a)
```

16.18 Definitions

1. *Higher-kinded polymorphism* is polymorphism that has a type variable abstracting over types of a higher kind. `Functor` is an example of higher-kinded polymorphism, because the kind of the `f` parameter to `Functor` is `* -> *`. Another example of higher-kinded polymorphism would be a datatype having a parameter to a type constructor that is of a higher kind, such as the following:

   ```
   data Weird f a = Weird (f a)
   ```

   Where the kinds of the types involved are:

   ```
   a     :: *
   f     :: * -> *
   Weird :: (* -> *) -> * -> *
   ```

   Here, both `Weird` and `f` are higher-kinded, with `Weird` being an example of higher-kinded polymorphism.

2. *Functor* is a mapping between categories. In Haskell, this manifests as a type class that generalizes the concept of `map`: it takes a function `(a -> b)` and lifts it into a different type. This conventionally implies some notion of a function that can be applied to a value with more structure than the unlifted function was originally designed for. The additional structure is represented by the use of a higher-kinded type `f`, introduced by the definition of the `Functor` type class:

[2]Thanks to Andraz Bajt for inspiring this exercise.

```
f       :: a -> b

-- "more structure"
fmap f :: f a -> f b

-- f is applied to a single argument
-- and so is kind * -> *
```

One should be careful not to confuse this intuition for it necessarily being exclusively about containers or data structures. There's a `Functor` of functions, and many exotic types will have a lawful `Functor` instance, as well.

3. Let's talk about *lifting*. Because most of the rest of the book deals with applicatives and monads of various flavors, we're going to be lifting a lot, but what do we mean? When Rudolf Carnap first described functors in the context of linguistics, he didn't really talk about it as lifting anything, and mathematicians have followed in his footsteps, focusing on mapping and the production of outputs from certain types of inputs. Very mathematical of them, and yet Haskellers use the lifting metaphor often (as we do, in this book).

   There are a couple of ways people commonly think about it. One is that we can lift a function into a context. Another is that we lift a function over some layer of structure to apply it. The effect is the same:

```
Prelude> fmap (+1) $ Just 1
Just 2
Prelude> fmap (+1) [1, 2, 3]
[2,3,4]
```

   In the first case, we lift the function into a `Maybe` context in order to apply it and, in the second case, into a list context. It can be helpful to think of it in terms of lifting the function into a context, because it's the context we've lifted the function into that determines how the function will get applied (to one value or, recursively, to many, for example). The context is the datatype, the definition of the datatype, and the `Functor` instance

we have for that datatype. It's also the contexts that determine what happens when we try to apply a function to an `a` that isn't there:

```
Prelude> fmap (+1) []
[]
Prelude> fmap (+1) Nothing
Nothing
```

But we often speak more casually about lifting over, as in `fmap` lifts a function *over* a data constructor. This works, too, if you think of the data constructor as a layer of structure. The function hops over that layer and applies to what's inside, if anything.

More precisely, lifting means applying a type constructor to a type, as in taking an `a` type variable and applying an `f` type constructor to it to get an `f a`. Keeping this definition in mind will be helpful. Remember to *follow the types* rather than getting too caught up in the web of a metaphor.

4. *George Clinton* is one of the most important innovators of funk music. Clinton headed up the bands Parliament and Funkadelic, whose collective style of music is known as P-Funk. The two bands fused into a single apotheosis of booty-shaking rhythm. The Parliament album *Mothership Connection* is one of the most famous and influential albums in rock history. Not a `Functor`, but you can pretend the album is mapping your consciousness from the real world into the category of funkiness, if that helps.

16.19 Follow-up resources

1. Haskell Wikibook. *The Functor class*.
`https://en.wikibooks.org/wiki/Haskell/The_Functor_class`
2. Mark P. Jones. *A system of constructor classes: overloading and implicit higher-order polymorphism*.
3. Gabriel Gonzalez. *The functor design pattern*.

Chapter 17

Applicative

> The images I most connect to, historically speaking, are in black and white. I see more in black and white—I like the abstraction of it.
>
> Mary Ellen Mark

17.1 Applicative

In the previous chapters, we saw two common algebras that are used as type classes. `Monoid` gives us a means of mashing two values of the same type together. `Functor`, on the other hand, is for function application *over* some structure we don't want to have to think about. `Monoid`'s core operation, `mappend`, smashes the structures together—when you `mappend` two lists, they become one list, so the structures themselves have been joined. However, the core operation of `Functor`, `fmap`, applies a function to a value that is within some structure, while leaving the structure itself unaltered.

We come now to `Applicative`. Applicatives are monoidal functors. No, no, stay with us. The `Applicative` type class allows for function application lifted over structure (like `Functor`). But with `Applicative`, the function we're applying is also embedded in some structure. Because the function *and* the value it's being applied to both have structure, we have to smash those structures together. So, `Applicative` involves monoids and functors. And that's a pretty powerful thing.

In this chapter, we will:

- Define and explore the `Applicative` type class and its core operations.
- Demonstrate why applicatives are monoidal functors.
- Make the usual chitchat about laws and instances.
- Do a lot of lifting.
- Give you some `Validation`.

17.2 Defining Applicative

The first thing you're going to notice about this type class declaration is that the `f` that represents the structure, similar to `Functor`, is itself constrained by the `Functor` type class:

```
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

So, every type that can have an `Applicative` instance must also have a `Functor` instance.

The `pure` function does a simple and very boring thing: it lifts something into functorial (applicative) structure. You can think of this as being a bare minimum bit of structure or structural *identity*. Identity for *what*, you'll see later when we go over the laws. The more interesting operation of this type class is `<*>`. This is an infix function called "apply" or sometimes "ap," or sometimes "tie fighter" when we're feeling particularly zippy.

If we compare the types of `<*>` and `fmap`, we see the similarity:

```
-- fmap
(<$>) :: Functor f
      =>   (a -> b) -> f a -> f b


(<*>) :: Applicative f
      => f (a -> b) -> f a -> f b
```

The difference is the `f` representing functorial structure that is on the outside of our function in the second definition. We'll see good examples of what that means in practice, in a moment.

Along with these core functions, the `Control.Applicative` library provides some other convenient functions: `liftA`, `liftA2`, and `liftA3`:

```
liftA  :: Applicative f =>
          (a -> b)
       -> f a
       -> f b

liftA2 :: Applicative f =>
          (a -> b -> c)
       -> f a
       -> f b
       -> f c
```

```
liftA3 :: Applicative f =>
          (a -> b -> c -> d)
       -> f a
       -> f b
       -> f c
       -> f d
```

If you're looking at the type of `liftA` and thinking, but that's `fmap`, you are correct. It is basically the same as `fmap` only with an `Applicative` type class constraint instead of a `Functor` one. Since all applicatives are also functors, though, this is a distinction without much significance.

Similarly, you can see that `liftA2` and `liftA3` are `fmap` but with functions involving more arguments. It can be a little difficult to wrap one's head around how those will work in practice, so we'll want to look next at some examples to start developing a sense of what applicatives can do for us.

17.3 Functor vs. Applicative

We've already said that applicatives are monoidal functors, so what we've already learned about `Monoid` and `Functor` is relevant to our understanding of `Applicative`. We've already seen some examples of what this means in practice, but we want to develop a stronger intuition for the relationship.

Let's review the difference between `fmap` and `<*>`:

```
fmap  ::   (a -> b) -> f a -> f b
(<*>) :: f (a -> b) -> f a -> f b
```

The difference, as you can see above, is that we now have an `f` in front of our function `(a -> b)`. The increase in power it introduces is profound. For one thing, any `Applicative` also has a `Functor` and not merely by definition—you can define a `Functor` in terms of a provided `Applicative` instance. Proving that is outside the scope of the current book, but this follows from the laws of `Functor` and `Applicative` (we'll get to the applicative laws later in this chapter):

```
fmap f x = pure f <*> x
```

How might we demonstrate this? You'll need to import the library `Control.Applicative` if you're using GHC 7.8 or older to test this example:

```
Prelude> fmap (+1) [1, 2, 3]
[2,3,4]

Prelude> pure (+1) <*> [1..3]
[2,3,4]
```

Keeping in mind that `pure` has type `Applicative f => a -> f a`, we can think of it as a means of embedding a value of any type in the structure we're working with:

```
Prelude> pure 1 :: [Int]
[1]
Prelude> pure 1 :: Maybe Int
Just 1
Prelude> pure 1 :: Either a Int
Right 1
Prelude> pure 1 :: ([a], Int)
([],1)
```

The left type is handled differently from the right in the final two examples for the same reason as here:

```
Prelude> fmap (+1) (4, 5)
(4,6)
```

The left type is part of the structure, and the structure is not transformed by the function application.

17.4 Applicative functors are monoidal functors

First, let us notice something:

```
 ($)  ::   (a -> b) ->   a ->   b
(<$>) ::   (a -> b) -> f a -> f b
(<*>) :: f (a -> b) -> f a -> f b
```

We already know `$` to be something of a do-nothing infix function that exists to give the right-hand side more precedence and thus avoid parentheses. For our present purposes, it acts as a nice proxy for ordinary function application in its type.

When we get to `<$>`, the alias for `fmap`, we notice that the first change is that we're now lifting our `(a -> b)` over the `f` wrapped around our value and applying the function to that value.

Then, as we arrive at `ap` or `<*>`, the `Applicative` apply method, our function is now also embedded in the functorial structure. Now we get to the *monoidal* in "monoidal functor":

```
:: f (a -> b) -> f a -> f b

-- The two arguments to our function are:

f (a -> b)
-- and
f  a
```

If we imagine that we can apply `(a -> b)` to `a` and get `b`, ignoring the functorial structure, we still have a problem, as we need to return `f b`. When we were dealing with `fmap`, we had only one bit of structure, so it was left unchanged. Now we have two bits of structure of type `f` that we need to deal with somehow before returning a value of type `f b`. We can't simply leave them unchanged; we must unite them somehow. They will be the same type, because the `f` must be the same type throughout. In fact, if we separate the *structure* parts from the *function* parts, maybe we'll see what we need:

```
:: f (a -> b) -> f a -> f b

   f            f      f
  (a -> b)      a      b
```

Didn't we have something earlier that can take two values of one type and return one value of the same type? Provided the `f` is a type with a `Monoid` instance, then we have a good way to make them play nice together:

```
mappend :: Monoid a => a -> a -> a
```

So, with `Applicative`, we have a `Monoid` for our structure and function application for our values!

```
mappend ::   f            f      f
$       ::  (a -> b)      a      b

(<*>) ::  f (a -> b)    -> f a -> f b

-- plus Functor fmap to be able to map
-- over the f to begin with
```

So, in a sense, we're bolting a `Monoid` onto a `Functor` to be able to deal with functions embedded in additional structure. In another sense, we're enriching function application with the very structure we were previously mapping over with `Functor`. Let's consider a few familiar examples to examine what this means, starting with list:

```
[(*2), (*3)] <*> [4, 5]
```

Equals:

```
[2 * 4, 2 * 5, 3 * 4, 3 * 5]
```

Reduced:

```
[8,10,12,15]
```

So what is `(a -> b)` enriched with in `f (a -> b) -> f a -> f b`? In this case, "list-ness." Although the actual application of each `(a -> b)` to a value of type `a` is quite ordinary, we now have a list of functions rather than a single function, as would be the case if it were the list `Functor`.

But lists aren't the only structure we can enrich our functions with—not even close! The structure bit can also be `Maybe`:

```
Just (*2) <*> Just 2
=
Just 4

Just (*2) <*> Nothing
=
Nothing

Nothing <*> Just 2
=
Nothing

Nothing <*> Nothing
=
Nothing
```

With `Maybe`, the ordinary functor is mapping over the possibility of a value's nonexistence. With the `Applicative`, now the function also might not be provided. We'll see a couple of nice, long examples of how this might happen—how you could end up not even providing a function to apply—in a bit, not just with `Maybe`, but with `Either` and a new type called `Validation`, as well.

Show me the monoids

Recall that the `Functor` instance for the 2-tuple ignores the first value inside the tuple:

```
Prelude> fmap (+1) ("blah", 0)
("blah",1)
```

But the `Applicative` for the 2-tuple demonstrates the monoid in `Applicative` nicely for us. In fact, if you call `:info` on `(,)` in your REPL, you'll notice something:

```
Prelude> :info (,)
data (,) a b = (,) a b
-- Defined in GHC.Tuple
...
```

```
instance Monoid a
      => Applicative ((,) a)
-- Defined in GHC.Base
...
instance (Monoid a, Monoid b)
       => Monoid (a, b)
```

For the `Applicative` instance of the 2-tuple, we don't need a `Monoid` for the `b`, because we're using function application to produce that `b`. However, for the first value in the tuple, we still need the `Monoid`, because we have two values and need to somehow turn that into one value of the same type:

```
Prelude> ("Woo", (+1)) <*> (" Hoo!", 0)
("Woo Hoo!", 1)
```

Notice that for the `a` value, we don't apply any function, but they combine themselves as if by magic; that's due to the `Monoid` instance for the `a` values. The function in the `b` position of the left tuple is applied to the value in the `b` position of the right tuple to produce a result. That function application is why we don't need a `Monoid` instance on the `b`.

Let's look at more such examples. Pay careful attention to how the `a` values in the tuples are combined:

```
Prelude> import Data.Monoid
Prelude> (Sum 2, (+1)) <*> (Sum 0, 0)
(Sum {getSum = 2},1)

Prelude> (Product 3, (+9))<*>(Product 2, 8)
(Product {getProduct = 6},17)

Prelude> (All True, (+1))<*>(All False, 0)
(All {getAll = False},1)
```

It doesn't really matter *what* `Monoid`, but we need some way of combining or choosing our values.

Tuple Monoid and Applicative side by side

Squint if you can't see it:

```
instance (Monoid a, Monoid b)
      => Monoid (a,b) where
    mempty = (mempty, mempty)
    (a, b) `mappend` (a',b') =
      (a `mappend` a', b `mappend` b')

instance Monoid a
      => Applicative ((,) a) where
    pure x = (mempty, x)
    (u, f) <*> (v, x) =
      (u `mappend` v, f x)
```

Maybe Monoid and Applicative

While applicatives are monoidal functors, be careful about making assumptions based on this. For one thing, `Monoid` and `Applicative` instances aren't required or guaranteed to have the same monoid of structure, and the functorial part may change the way it behaves. Nevertheless, you might be able to see the implicit monoid in how the `Applicative` pattern matches on the `Just` and `Nothing` cases and compare that with this `Monoid`:

```
instance Monoid a => Monoid (Maybe a) where
  mempty = Nothing
  mappend m Nothing = m
  mappend Nothing m = m
  mappend (Just a) (Just a') =
    Just (mappend a a')

instance Applicative Maybe where
    pure = Just

    Nothing <*> _     = Nothing
    _ <*> Nothing     = Nothing
    Just f <*> Just a = Just (f a)
```

Later, we'll see some examples of how different `Monoid` instances can give different results for applicatives. For now, recognize that the monoidal bit may not be what you recognize as the canonical `mappend` of that type, because some types can have multiple monoids.

17.5 Applicative in use

By now, it should come as no surprise that many of the datatypes we've been working with in the past two chapters also have `Applicative` instances. Since we are already so familiar with list and `Maybe`, those examples will be a good place to start. Later in the chapter, we will be introducing some new types, so just hang onto your hats.

List Applicative

We'll start with the list `Applicative`, because it's a clear way to get a sense of the pattern. Let's start by specializing the types:

```
-- f ~ []

(<*>) ::  f  (a -> b)  ->  f  a ->  f  b
(<*>) :: [ ] (a -> b)  -> [ ] a -> [ ] b

-- more syntactically typical
(<*>) ::    [(a -> b)] -> [a]   ->    [b]

pure :: a ->   f  a
pure :: a ->  [ ] a
```

Or, again, if you have GHC 8 or newer, you can do this:

```
Prelude> :set -XTypeApplications
Prelude> :type (<*>) @[]
(<*>) @[] :: [a -> b] -> [a] -> [b]
Prelude> :type pure @[]
pure @[] :: a -> [a]
```

What does the list Applicative do?

Previously, with the list `Functor`, we mapped a single function over a plurality of values:

```
Prelude> fmap (2^) [1, 2, 3]
[2,4,8]
Prelude> fmap (^2) [1, 2, 3]
[1,4,9]
```

With the list `Applicative`, we can map a plurality of functions over a plurality of values:

```
Prelude> [(+1), (*2)] <*> [2, 4]
[3,5,4,8]
```

We can see how this makes sense given that:

```
(<*>) :: Applicative f
      => f (a -> b) -> f a -> f b

f ~ []

listApply :: [(a -> b)] -> [a] -> [b]

listFmap  ::  (a -> b)  -> [a] -> [b]
```

The `f` structure that is wrapped around our function in the `listApply` function is itself a list. Therefore, our `a -> b` from `Functor` has become a *list* of `a -> b`.

Now, what happens with that expression we were testing? Something like this:

```
[(+1), (*2)] <*> [2, 4] == [3,5,4,8]

    [ 3   ,   5   ,   4   ,   8 ]
--   [1]     [2]     [3]     [4]
```

1. The first item in the list, 3, is the result of `(+1)` being applied to 2.

2. 5 is the result of applying `(+1)` to 4.
3. 4 is the result of applying `(*2)` to 2.
4. 8 is the result of applying `(*2)` to 4.

More visually:

```
[(+1), (*2)] <*> [2, 4]
```

```
[ (+1) 2 , (+1) 4 , (*2) 2 , (*2) 4 ]
```

It maps each function value from the first list over the second list, applies the operations, and returns one list. The fact that it doesn't return two lists or a nested list or some other configuration in which both structures are preserved is the monoidal part. The reason we don't have a list of functions concatenated with a list of values is the function application part.

We can see this relationship more readily if we use the tuple constructor with the list `Applicative`. We'll use the infix operator for `fmap` to map the tuple constructor over the first list. This embeds an unapplied function (the tuple data constructor, in this case) into some structure (a list, in this case), and returns a list of partially applied functions. The (infix) applicative will then apply one list of operations to the second list, monoidally appending the two lists:

```
Prelude> (,) <$> [1, 2] <*> [3, 4]
[(1,3),(1,4),(2,3),(2,4)]
```

You might think of it this way:

```
Prelude> (,) <$> [1, 2] <*> [3, 4]
```

`fmap` the `(,)` over the first list:

```
[(1, ), (2, )] <*> [3, 4]
```

Then, we apply the first list to the second:

```
[(1,3),(1,4),(2,3),(2,4)]
```

The `liftA2` function gives us another way to write this, too:

```
Prelude> liftA2 (,) [1, 2] [3, 4]
[(1,3),(1,4),(2,3),(2,4)]
```

Let's look at a few more examples of the same pattern:

```
Prelude> (+) <$> [1, 2] <*> [3, 5]
[4,6,5,7]
Prelude> liftA2 (+) [1, 2] [3, 5]
[4,6,5,7]

Prelude> max <$> [1, 2] <*> [1, 4]
[1,4,2,4]
Prelude> liftA2 max [1, 2] [1, 4]
[1,4,2,4]
```

If you're familiar with Cartesian products,[1] this probably looks a lot like one but with functions.

We're going to run through some more examples, to give you a little more context for when these functions can be useful. The following examples will use a function called `lookup` that we'll briefly demonstrate:

```
Prelude> :t lookup
lookup :: Eq a => a -> [(a, b)] -> Maybe b
Prelude> l = lookup 3 [(3, "hello")]
Prelude> l
Just "hello"
Prelude> fmap length $ l
Just 5
Prelude> c (x:xs) = toUpper x:xs
Prelude> fmap c $ l
Just "Hello"
```

So, `lookup` searches inside a list of tuples for a value that matches the input and returns the paired value wrapped inside a `Maybe` context.

It's worth pointing out here that if you're working with `Map` data structures instead of lists of tuples, you can import `Data.Map` and use

[1]The Cartesian product is the product of two sets that results in all the ordered pairs (tuples) of the elements of those sets.

a `Map` version of `lookup` along with `fromList` to accomplish the same thing with that data structure:

```
Prelude> m = fromList [(3, "hello")]
Prelude> fmap c $ Data.Map.lookup 3 m
Just "Hello"
```

That may seem trivial at the moment, but `Map` is a frequently used data structure, so it's worth mentioning.

Now that we have values wrapped in a `Maybe` context, perhaps we'd like to apply some functions to them. This is where we want applicative operations. Although it's more likely that we'd have functions fetching data from somewhere else rather than having it all listed in our code, we'll go ahead and define some values in a source file for convenience:

```
import Control.Applicative

f x =
  lookup x [ (3, "hello")
           , (4, "julie")
           , (5, "kbai")]

g y =
  lookup y [ (7, "sup?")
           , (8, "chris")
           , (9, "aloha")]

h z =
  lookup z [(2, 3), (5, 6), (7, 8)]
m x =
  lookup x [(4, 10), (8, 13), (1, 9001)]
```

Now, we want to look things up and add them together. We'll start with some operations over these data:

```
Prelude> f 3
Just "hello"
Prelude> g 8
Just "chris"
```

```
Prelude> (++) <$> f 3 <*> g 7
Just "hellosup?"
Prelude> (+) <$> h 5 <*> m 1
Just 9007
Prelude> (+) <$> h 5 <*> m 6
Nothing
```

So, we first `fmap` those functions over the value inside the first `Maybe` context, if it's a `Just` value, making it a partially applied function wrapped in a `Maybe` context. Then, we use the tie-fighter to apply that to the second value, again wrapped in a `Maybe`. If either value is a `Nothing`, we get `Nothing`.

We can again do the same thing with `liftA2`:

```
Prelude> liftA2 (++) (g 9) (f 4)
Just "alohajulie"
Prelude> liftA2 (^) (h 5) (m 4)
Just 60466176
Prelude> liftA2 (*) (h 5) (m 4)
Just 60
Prelude> liftA2 (*) (h 1) (m 1)
Nothing
```

Your applicative context can also sometimes be `IO`:

```
(++) <$> getLine <*> getLine
(,)  <$> getLine <*> getLine
```

Try it. Now try using `fmap` to get the length of the string result from the first example.

Exercises: Lookups

In the following exercises, you will need to use the given terms to make the expressions type check:

1. `pure`
2. `(<$>)`

```
-- or fmap
```

3. `(<*>)`

Make the following expressions type check:

1. ```
   added :: Maybe Integer
   added =
     (+3) (lookup 3 $ zip [1, 2, 3] [4, 5, 6])
   ```

2. ```
   y :: Maybe Integer
   y = lookup 3 $ zip [1, 2, 3] [4, 5, 6]

   z :: Maybe Integer
   z = lookup 2 $ zip [1, 2, 3] [4, 5, 6]

   tupled :: Maybe (Integer, Integer)
   tupled = (,) y z
   ```

3. ```
   import Data.List (elemIndex)

   x :: Maybe Int
   x = elemIndex 3 [1, 2, 3, 4, 5]

   y :: Maybe Int
   y = elemIndex 4 [1, 2, 3, 4, 5]

   max' :: Int -> Int -> Int
   max' = max

   maxed :: Maybe Int
   maxed = max' x y
   ```

4. ```
   xs = [1, 2, 3]
   ys = [4, 5, 6]

   x :: Maybe Integer
   x = lookup 3 $ zip xs ys

   y :: Maybe Integer
   y = lookup 2 $ zip xs ys

   summed :: Maybe Integer
   summed = sum $ (,) x y
   ```

Identity

The `Identity` type here is a way to introduce structure without changing the semantics of what you're doing. We'll see it used with type classes that involve function application around and over structure, but this type itself isn't very interesting, as it has no semantic flavor.

Specializing the types

Here is what the type will look like when our structure is `Identity`:

```
-- f ~ Identity
-- Applicative f =>
type Id = Identity

(<*>) ::  f (a -> b) ->  f a ->  f b
(<*>) :: Id (a -> b) -> Id a -> Id b

pure :: a ->  f a
pure :: a -> Id a
```

Why would we use `Identity` just to introduce some structure? What is the meaning of all this?

```
Prelude> xs = [1, 2, 3]
Prelude> xs' = [9, 9, 9]
Prelude> const <$> xs <*> xs'
[1,1,1,2,2,2,3,3,3]
Prelude> mkId = Identity
Prelude> const <$> mkId xs <*> mkId xs'
Identity [1,2,3]
```

Having this extra bit of structure around our values lifts the `const` function, from mapping over the lists to mapping over the `Identity`. We have to go over an `f` structure to apply the function to the values inside. If our `f` is the list, `const` applies to the values inside the list, as we saw above. If the `f` is `Identity`, then `const` treats the lists inside the `Identity` structure as single values, not structure-containing values.

Exercise: Identity instance

Write an `Applicative` instance for `Identity`:

```
newtype Identity a = Identity a
    deriving (Eq, Ord, Show)

instance Functor Identity where
  fmap = undefined

instance Applicative Identity where
  pure = undefined
  (<*>) = undefined
```

Constant

This is not so different from the `Identity` type, except it not only provides structure, it also acts like the `const` function. It sort of throws away a function application. If this seems confusing, it's because it is. However, it is also something that, like `Identity`, has real use cases, and you will see it in other people's code. It can be difficult to get used to using it yourself, but we keep trying.

This datatype is like the `const` function, in that it takes two arguments, but one of them gets discarded. In the case of the datatype, we have to map our function over the argument that gets discarded. So there is no value to map over, and the function application doesn't happen.

Specializing the types

All right, so here's what the types will look like:

```
newtype Constant a b =
  Constant { getConstant :: a }

-- f ~ Constant e
type C = Constant
```

```
(<*>) ::   f (a -> b) ->   f a ->   f b
(<*>) :: C e (a -> b) -> C e a -> C e b

pure :: a ->   f a
pure :: a -> C e a
```

And here are some examples of how it works. These are, yes, a bit contrived, but showing you *real code* with this in it would probably make it much harder for you to see what's going on:

```
Prelude> f = Constant (Sum 1)
Prelude> g = Constant (Sum 2)
Prelude> f <*> g
Constant {getConstant = Sum {getSum = 3}
Prelude> Constant undefined <*> g
Constant (Sum {getSum =
  *** Exception: Prelude.undefined
Prelude> pure 1
1
Prelude> pure 1 :: Constant String Int
Constant {getConstant = ""}
```

It can't do anything, because it can only hold onto the one value. The function doesn't exist, and the `b` is a ghost. So, you use this when whatever you want to do involves throwing away a function application. We know it seems somewhat crazy, but we promise there are really times when real coders do this in real code. Pinky swear.

Exercise: Constant instance

Write an `Applicative` instance for `Constant`:

```
newtype Constant a b =
  Constant { getConstant :: a }
  deriving (Eq, Ord, Show)

instance Functor (Constant a) where
  fmap = undefined
```

```
instance Monoid a
      => Applicative (Constant a) where
  pure = undefined
  (<*>) = undefined
```

Maybe Applicative

With `Maybe`, we're doing something a bit different from the above. We saw previously how to use `fmap` with `Maybe`, but here our function is also embedded in a `Maybe` structure. Therefore, when `f` is `Maybe`, we're saying the function itself might not exist, because we're allowing the possibility that the function to be applied is a `Nothing` case.

Specializing the types

Here's what the type looks like when we're using `Maybe` as our `f` structure:

```
-- f ~ Maybe
type M = Maybe

(<*>) :: f (a -> b) -> f a -> f b
(<*>) :: M (a -> b) -> M a -> M b

pure :: a -> f a
pure :: a -> M a
```

Are you ready to validate some persons? Yes. Yes, you are.

Using the Maybe Applicative

Consider the following example, where we validate our inputs to create a value of type `Maybe Person`. We use `Maybe`, because our inputs might be invalid:

```
validateLength :: Int
               -> String
               -> Maybe String
validateLength maxLen s =
  if (length s) > maxLen
  then Nothing
  else Just s
```

```
newtype Name =
  Name String deriving (Eq, Show)
newtype Address =
  Address String deriving (Eq, Show)

mkName :: String -> Maybe Name
mkName s =
  fmap Name $ validateLength 25 s

mkAddress :: String -> Maybe Address
mkAddress a =
  fmap Address $ validateLength 100 a
```

Now, we'll make a smart constructor for a `Person`:

```
data Person =
  Person Name Address
  deriving (Eq, Show)

mkPerson :: String
         -> String
         -> Maybe Person

mkPerson n a =
  case mkName n of
   Nothing -> Nothing
   Just n' ->
     case mkAddress a of
      Nothing -> Nothing
      Just a' ->
        Just $ Person n' a'
```

The problem here is while we've successfully leveraged `fmap` from `Functor` in the simpler cases of `mkName` and `mkAddress`, we can't really make that work with `mkPerson`. Let's investigate why:

```
Prelude> :t fmap Person (mkName "Babe")
fmap Person (mkName "Babe")
  :: Maybe (Address -> Person)
```

This has worked so far for the first argument to the `Person` constructor that we're validating, but we've hit a roadblock. Can you see the problem?

```
Prelude> :{
*Main| fmap (fmap Person (mkName "Babe"))
*Main|      (mkAddress "old macdonald's")
*Main| :}

• Couldn't match expected type
  ‘Address -> b’
              with actual type
                ‘Maybe (Address -> Person)’
• Possible cause: ‘fmap’ is applied to too
    many arguments
  In the first argument of ‘fmap’, namely
    ‘(fmap Person (mkName "Babe"))’
  In the expression:
    fmap (fmap Person (mkName "Babe"))
      (mkAddress "old macdonald's")
  In an equation for ‘it’:
      it
        = fmap (fmap Person (mkName "Babe"))
            (mkAddress "old macdonald's")
• Relevant bindings include
    it :: Maybe b
```

The problem is that our `(a -> b)` is now hiding inside `Maybe`. Let's look at the type of `fmap` again:

```
fmap :: Functor f => (a -> b) -> f a -> f b
```

`Maybe` is definitely a `Functor`, but that's not really going to help us here. We need to be able to map a function embedded in our `f`. `Applicative` gives us what we need here!

```
(<*>) :: Applicative f
      => f (a -> b) -> f a -> f b
```

Now, let's see if we can wield this new toy:

```
Prelude> s = "old macdonald's"
Prelude> addy = mkAddress s
Prelude> b = mkName "Babe"
Prelude> person = fmap Person b
Prelude> person <*> addy
Just (Person (Name "Babe")
             (Address "old macdonald's"))
```

Nice, right? A little ugly, though. Using the infix alias for `fmap` called `<$>` cleans it up a bit, at least to Haskellers' eyes:

```
Prelude> Person <$> mkName "Babe" <*> addy
Just (Person (Name "Babe")
             (Address "old macdonald's"))
```

We still use `fmap` (via `<$>`) here for the first lifting over `Maybe`. After that, our `(a -> b)` is hiding in the `f` where `f = Maybe`, so we have to start using `Applicative` to keep mapping over it.

We can now use a much shorter definition of `mkPerson`!

```
mkPerson :: String
         -> String
         -> Maybe Person
mkPerson n a =
  Person <$> mkName n <*> mkAddress a
```

As an additional bonus, this is now far less annoying to extend if we need to add new fields, as well.

Breaking down that example

We're going to give the `Functor` and `Applicative` instances for the `Maybe` type the same treatment that we gave to folds. This will be a bit long. It is possible that some of this will seem like far too much detail—read it to whatever depth you feel you need at this time. It will sit here, patiently waiting to see if you ever decide to come back and go over it more closely.

Maybe Functor and the Name constructor

```
instance Functor Maybe where
    fmap _ Nothing    = Nothing
    fmap f (Just a)   = Just (f a)

instance Applicative Maybe where
    pure = Just

    Nothing <*> _     = Nothing
    _ <*> Nothing     = Nothing
    Just f <*> Just a = Just (f a)
```

The `Applicative` instance is not exactly the same as the instance in `base`, but that's for simplification. For your purposes, it produces the same results.

First, the function and datatype definitions for our functor write-up for how we're using the `validateLength` function with `Name` and `Address`:

```
validateLength :: Int
               -> String
               -> Maybe String
validateLength maxLen s =
  if (length s) > maxLen
  then Nothing
  else Just s

newtype Name =
  Name String deriving (Eq, Show)

newtype Address =
  Address String deriving (Eq, Show)

mkName :: String -> Maybe Name
mkName s = fmap Name $ validateLength 25 s

mkAddress :: String -> Maybe Address
mkAddress a =
  fmap Address $ validateLength 100 a
```

Now, we're going to start filling in the definitions and expanding them equationally like we did in Chapter 10.

First we apply `mkName` to the value `"babe"` so that `s` is bound to that string:

```
mkName s =
  fmap Name $ validateLength 25 s
mkName "babe" =
  fmap Name $ validateLength 25 "babe"
```

Now, we need to figure out what `validateLength` is about, since that has to be evaluated before we know what `fmap` is mapping over. Here, we're applying it to 25 and `"babe"`, evaluating the length of the string `"babe"` and then determining which branch in the if-then-else expression wins:

```
validateLength :: Int
               -> String
               -> Maybe String

validateLength 25 "babe" =
  if (length "babe") > 25
  then Nothing
  else Just "babe"

  if 4 > 25
  then Nothing
  else Just "babe"

-- 4 isn't greater than 25, so:
validateLength 25 "babe" =
  Just "babe"
```

Next, we're going to replace `validateLength` applied to 25 and `"babe"` with what it evaluates to, then figure out what the `fmap Name` over `Just "babe"` business is about:

```
mkName "babe" =
  fmap Name $ Just "babe"


fmap Name $ Just "babe"
```

Keeping in mind the type of `fmap` from `Functor`, we see the data constructor `Name` is the function `(a -> b)` we're mapping over some functorial `f`. In this case, `f` is `Maybe`. The `a` in `f a` is type `String`:

```
(a -> b) -> f a -> f b
```

```
:t Name        :: (String -> Name)
:t Just "babe" :: Maybe String
```

```
type M = Maybe
```

```
    (a -> b)    -> f a      -> f b
(String -> Name) -> M String -> M Name
```

Since we know we're dealing with the `Functor` instance for `Maybe`, we can inline *that* function's definition, too!

```
fmap _ Nothing  = Nothing
fmap f (Just a) = Just (f a)
```

```
-- We have (Just "babe") so
-- skipping Nothing case
-- fmap _ Nothing = Nothing
```

```
fmap f    (Just a)      =
  Just (f a)
fmap Name (Just "babe") =
  Just (Name "babe")
```

```
mkName "babe" = fmap Name $ Just "babe"
mkName "babe" = Just (Name "babe")
--              f     b
```

Maybe Applicative and Person

```
data Person =
  Person Name Address
  deriving (Eq, Show)
```

First, we'll be using the `Functor` to map the `Person` data constructor over the `Maybe Name` value. Unlike `Name` and `Address`, `Person` takes two arguments rather than one:

```
    Person
<$> Just (Name "babe")
<*> Just (Address "farm")

fmap Person (Just (Name "babe"))

:t Person :: Name -> Address -> Person

:t Just (Name "babe") :: Maybe Name

(a -> b) -> f a -> f b
  (Name -> Address -> Person)
     a    -> b
-> Maybe Name -> Maybe (Address -> Person)
   f     a         f             b

fmap _ Nothing       = Nothing
fmap f (Just a)      = Just (f a)

fmap Person (Just (Name "babe"))

f :: Person
a :: Name "babe"

-- We skip this pattern match
-- because we have
-- Just fmap _ Nothing = Nothing

fmap f      (Just a)            =
  Just (f a)

fmap Person (Just (Name "babe")) =
  Just (Person (Name "babe"))
```

The problem is that `Person (Name "babe")` is awaiting another argument, the address, so it's partially applied. That's our `(a -> b)` in the type of the `Applicative` apply function, `<*>`. The `f` wrapping our function `(a -> b)` is the `Maybe` that results from us possibly not having had an `a` to map over to begin with, resulting in a `Nothing` value:

```
-- Person is awaiting another argument
:t Just (Person (Name "babe"))
     :: Maybe (Address -> Person)

:t Just (Address "farm") :: Maybe Address

-- We want to apply the partially
-- applied (Person "babe") inside the
-- Just to the "farm" inside the Just.

    Just (Person (Name "babe"))
<*> Just (Address "farm")
```

So, since the function we want to map is inside the same structure as the value we want to apply it to, we need the `Applicative` `(<*>)`. In the following, we remind you of what the type looks like and how the type specializes to this application:

```
f (a -> b) -> f a -> f b

type M = Maybe
type Addy = Address

M (Addy -> Person) -> M Addy -> M Person
f (  a  ->   b   ) -> f a    -> f b
```

We know we're using the `Maybe` `Applicative`, so we can go ahead and inline the definition. Reminder that this version of the `Applicative` instance is simplified from the one in GHC, so please don't email us to tell us our instance is wrong:

```
instance Applicative Maybe where
    pure = Just

    Nothing <*> _      = Nothing
    _ <*> Nothing      = Nothing
    Just f <*> Just a = Just (f a)
```

We know we can ignore the `Nothing` cases, because our function is `Just`, our value is `Just`, and... our cause is just! Just... kidding.

If we fill in our partially applied `Person` constructor for `f`, and our `Address` value for `a`, it's not too hard to see how the final result fits:

```
-- Neither function nor value are Nothing,
-- so we skip these two cases
-- Nothing <*> _ = Nothing
-- _ <*> Nothing = Nothing

Just  f <*> Just a = Just (f a)
Just (Person (Name "babe"))
  <*> Just (Address "farm") =
    Just (Person (Name "babe")
                 (Address "farm"))
```

Before we moooove on

```
data Cow = Cow {
      name   :: String
    , age    :: Int
    , weight :: Int
   } deriving (Eq, Show)

noEmpty :: String -> Maybe String
noEmpty ""  = Nothing
noEmpty str = Just str

noNegative :: Int -> Maybe Int
noNegative n | n >= 0 = Just n
             | otherwise = Nothing

-- Validating to get rid of empty
-- strings and negative numbers
cowFromString :: String
              -> Int
              -> Int
              -> Maybe Cow

cowFromString name' age' weight' =
  case noEmpty name' of
   Nothing -> Nothing
```

```
    Just nammy ->
     case noNegative age' of
      Nothing -> Nothing

      Just agey ->
       case noNegative weight' of
        Nothing -> Nothing

        Just weighty ->
          Just (Cow nammy agey weighty)
```

`cowFromString` is... bad. You can probably tell. But by the use of `Applicative`, it can be improved!

```
cowFromString' :: String
               -> Int
               -> Int
               -> Maybe Cow
cowFromString' name' age' weight' =
  Cow <$> noEmpty name'
      <*> noNegative age'
      <*> noNegative weight'
```

Or, if we want other Haskellers to think we're really cool and hip:

```
cowFromString'' :: String
                -> Int
                -> Int
                -> Maybe Cow
cowFromString'' name' age' weight' =
  liftA3 Cow (noEmpty name')
             (noNegative age')
             (noNegative weight')
```

So, we're taking advantage of the `Maybe Applicative` here. What does that look like? First, we'll use the infix syntax for `fmap`, `<$>`, and apply, `<*>`:

```
Prelude> cow1 = Cow <$> noEmpty "Bess"
```

```
Prelude> :t cow1
cow1 :: Maybe (Int -> Int -> Cow)

Prelude> cow2 = cow1 <*> noNegative 1

Prelude> :t cow2
cow2 :: Maybe (Int -> Cow)

Prelude> cow3 = cow2 <*> noNegative 2

Prelude> :t cow3
cow3 :: Maybe Cow
```

Then with `liftA3`:

```
Prelude> cow1 = liftA3 Cow

Prelude> :t cow1
cow1 :: Applicative f
     => f String -> f Int -> f Int -> f Cow

Prelude> cow2 = cow1 (noEmpty "blah")

Prelude> :t cow2
cow2 :: Maybe Int -> Maybe Int -> Maybe Cow

Prelude> cow3 = cow2 (noNegative 1)

Prelude> :t cow3
cow3 :: Maybe Int -> Maybe Cow

Prelude> cow4 = cow3 (noNegative 2)

Prelude> :t cow4
cow4 :: Maybe Cow
```

So, from a simplified point of view, `Applicative` is really just a way of saying:

```
-- we fmap'd my function over some
-- functorial f or it already
-- was in f somehow

-- f ~ Maybe
cow1 :: Maybe (Int -> Int -> Cow)
cow1 = fmap Cow (noEmpty "Bess")

-- and we hit a situation where want to map
--           f (a -> b)
-- not just    (a -> b)

(<*>) :: Applicative f
      => f (a -> b) -> f a -> f b
-- over some   f a
-- to get an   f b

cow2 :: Maybe (Int -> Cow)
cow2 = cow1 <*> noNegative 1
```

As a result, you may be able to imagine yourself saying, "I want to do something kinda like an `fmap`, but my function is embedded in the functorial structure too, not only the value I want to apply my function to." This is a basic motivation for `Applicative`.

With the `Applicative` instance for `Maybe`, what we're doing is enriching functorial application with the additional proviso that, "I may not have a function at all."

We can see this in the following specialization of the apply function, or `<*>`:

```
(<*>) :: Applicative f
      => f (a -> b) -> f a -> f b

f ~ Maybe
type M = Maybe
maybeApply :: M (a -> b) -> M a -> M b
maybeFmap  ::   (a -> b) -> M a -> M b

-- maybeFmap is just fmap's type
-- specialized to Maybe
```

You can test these specializations (more concrete versions) of the types:

```
maybeApply :: Maybe (a -> b)
           -> Maybe a
           -> Maybe b
maybeApply = (<*>)

maybeMap :: (a -> b)
         -> Maybe a
         -> Maybe b
maybeMap = fmap
```

If you make any mistakes, the compiler will let you know:

```
maybeMapBad :: (a -> b)
            -> Maybe a
            -> f b
maybeMapBad = fmap
```

```
• Couldn't match type ‘f’ with ‘Maybe’
  ‘f’ is a rigid type variable bound by
    the type signature for:
      maybeMapBad :: forall a b
        (f :: * -> *). (a -> b) ->
          Maybe a -> f b
```

Exercise: Fixer upper

Given the functions and values provided, use `<$>` from `Functor` and `<*>` and `pure` from the `Applicative` type class to fill in missing bits of the broken code below to make it work:

1.
```
const <$> Just "Hello" <*> "World"
```

2.
```
    (,,,) Just 90
  <*> Just 10 Just "Tierness" [1, 2, 3]
```

17.6 Applicative laws

After examining the law, test each of the expressions in the REPL.

1. Identity

   Here is the definition of the identity law:

   ```
   pure id <*> v = v
   ```

   To see examples of this law, evaluate these expressions:

   ```
   pure id <*> [1..5]

   pure id <*> Just "Hello Applicative"

   pure id <*> Nothing

   pure id <*> Left "Error'ish"

   pure id <*> Right 8001

   -- ((->) a) has an instance
   pure id <*> (+1) $ 2
   ```

   As you may recall, `Functor` has a similar identity law, and comparing them directly might help you see what's happening:

   ```
   id [1..5]

   fmap id [1..5]

   pure id <*> [1..5]
   ```

   The identity law states that all three of those should be equal. You can test them for equality in your REPL, or you could write a simple test to get the answer. So, what's `pure` doing for us? It's embedding our `id` function into some structure, so that we can use `apply` instead of `fmap`.

2. Composition

   Here is the definition of the composition law for applicatives:

```
pure (.) <*> u <*> v <*> w =
  u <*> (v <*> w)
```

You may find the syntax a bit unusual and difficult to read here. This is similar to the law of composition for `Functor`. It is the law stating that the result of composing our functions first and then applying them and the result of applying the functions first and then composing them should be the same. We're using the composition operator as a prefix instead of the more usual infix, and we're using `pure` in order to embed that operator into the appropriate structure, so that it can work with `apply`:

```
    pure (.)
<*> [(+1)]
<*> [(*2)]
<*> [1, 2, 3]

[(+1)] <*> ([(*2)] <*> [1, 2, 3])
```

```
    pure (.)
<*> Just (+1)
<*> Just (*2)
<*> Just 1

     Just (+1)
<*> (Just (*2) <*> Just 1)
```

This law is meant to ensure that there are no surprises resulting from composing your function applications.

3. Homomorphism

   A *homomorphism* is a structure-preserving map between two algebraic structures. The effect of applying a function that is embedded in some structure to a value that is embedded in some structure should be the same as applying a function to a value without affecting any outside structure:

   ```
   pure f <*> pure x = pure (f x)
   ```

   That's the statement of the law. Here's how it looks in practice:

```
pure (+1) <*> pure 1
```

```
pure ((+1) 1)
```

Those two lines of code should give you the same result. In fact, the result you see for those should be indistinguishable from the result of:

```
(+1) 1
```

The structure that `pure` is providing isn't meaningful. So, you can think of this law as having to do with the monoidal part of the applicative deal: the result should be the result of the function application without doing anything other than combining the structure bits. Just as we saw how `fmap` is really just a special type of function application that ignores a context, or surrounding structure, applicative is also function application that preserves structure. However, with applicative, since the function being applied *also* has structure, the structures have to be monoidal and come together in some fashion:

```
pure (+1) <*> pure 1 :: Maybe Int
```

```
pure ((+1) 1) :: Maybe Int
```

Those two results should, again, be the same, but this time, the structure is being provided by `Maybe`, so will the result of:

```
(+1) 1
```

Be equal this time around?

Here are a couple more examples to try out:

```
pure (+1) <*> pure 1 :: [Int]
```

```
pure (+1) <*> pure 1 :: Either a Int
```

The general idea of the homomorphism law is that applying the function doesn't change the structure around the values.

4. Interchange

   We begin again by looking at the definition of the interchange law:

   ```
   u <*> pure y = pure ($ y) <*> u
   ```

   It might help to break that down a bit. To the left of `<*>` must always be a function embedded in some structure. In the above definition, `u` represents a function embedded in some structure:

   ```
   Just (+2) <*> pure 2
   --  u     <*> pure y
   -- equals
   Just 4
   ```

   The right side of the definition might be a bit less obvious. By sectioning the `$` function application operator with the `y`, we create an environment in which the `y` is there, awaiting a function to apply to it. Let's try lining up the types again and see if that clears this up:

   ```
   pure ($ 2) <*> Just (+ 2)


   -- Remember, ($ 2) can become more concrete
        ($ 2) :: Num a => (a -> b) -> b
   Just (+ 2) :: Num a => Maybe (a -> a)
   ```

   If you're a bit confused by `($ 2)`, keep in mind that this is sectioning the dollar-sign operator and applying the second argument only, not the first. As a result, the type changes in the following manner:

   ```
   -- These are the same
   ($ 2)
   \f -> f $ 2


   ($)   :: (a -> b) -> a -> b
   ($ 2) :: (a -> b)      -> b
   ```

   Then, concretizing the types of the `Applicative` methods:

```
mPure :: a -> Maybe a
mPure = pure

embed :: Num a => Maybe ((a -> b) -> b)
embed = mPure ($ 2)

mApply :: Maybe ((a -> b) -> b)
       -> Maybe  (a -> b)
       -> Maybe             b
mApply = (<*>)

myResult = pure ($ 2) `mApply` Just (+2)
-- myResult == Just 4
```

Then, translating the types side by side, with different letters for some of the type variables to avoid confusion when comparing the original types with their more concrete forms:

```
(<*>) :: Applicative f
      => f (x -> y)
      -> f x
      -> f y

mApply :: Maybe ((a -> b) -> b)
       -> Maybe  (a -> b)
       -> Maybe             b

f           ~  Maybe
x           ~  (a -> b)
y           ~             b
(x -> y)    ~  (a -> b) -> b
```

According to the interchange law, this should be true:

```
    (Just (+2)  <*> pure 2)
 == (pure ($ 2) <*> Just (+2))
```

And you can see why that should be true, because despite the weird syntax, the two functions are doing the same job. Here are some more examples for you to try out:

```
[(+1), (*2)] <*> pure 1

pure ($ 1) <*> [(+1), (*2)]

Just (+3) <*> pure 1

pure ($ 1) <*> Just (+3)
```

Every `Applicative` instance you write should obey these four laws. This keeps your code composable and free of unpleasant surprises.

17.7 You knew this was coming

Property testing the applicative laws! You should have gotetn the gist of how to write property tests based on laws, so we're going to use a library this time. Conal Elliott has a nice library called `checkers` on Hackage and GitHub that provides some nice properties and utilities for `QuickCheck`.

After installing `checkers`, we can reuse the existing properties for validating `Monoid` and `Functor` instances, to revisit what we did previously:

```
module BadMonoid where

import Data.Monoid
import Test.QuickCheck
import Test.QuickCheck.Checkers
import Test.QuickCheck.Classes

data Bull =
    Fools
  | Twoo
  deriving (Eq, Show)

instance Arbitrary Bull where
  arbitrary =
    frequency [ (1, return Fools)
              , (1, return Twoo) ]
```

```
instance Monoid Bull where
  mempty = Fools
  mappend _ _ = Fools

-- EqProp is from the checkers library
instance EqProp Bull where
  (=-=) = eq

main :: IO ()
main = quickBatch (monoid Twoo)
```

There are some differences here worth noting. One is that we don't have to define the `Monoid` laws as `QuickCheck` properties ourselves; they are already bundled into a `TestBatch` called `monoid`. Another is that we need to define `EqProp` for our custom datatype. This is straightforward, because `checkers` exports a function called `eq` that reuses the pre-existing `Eq` instance for the datatype. Finally, we're passing a value of our type to `monoid` so it knows which `Arbitrary` instance to use to get random values—note that it doesn't *use* this value for anything.

Then, we can run `main` to kick it off and see how it goes:

```
Prelude> main

monoid:
  left  identity:
   *** Failed! Falsifiable (after 1 test):
Twoo
  right identity:
   *** Failed! Falsifiable (after 2 tests):
Twoo
  associativity:  +++ OK, passed 500 tests.
```

As we expect, it is able to falsify left and right identity for `Bull`. Now, let's test a pre-existing `Applicative` instance, such as list or `Maybe`. The type for the `TestBatch` that validates `Applicative` instances is a bit gnarly, so please bear with us:

```
applicative
:: ( Show a, Show (m a), Show (m (a -> b))
   , Show (m (b -> c)), Applicative m
   , CoArbitrary a, EqProp (m a)
   , EqProp (m b), EqProp (m c)
   , Arbitrary a, Arbitrary b
   , Arbitrary (m a)
   , Arbitrary (m (a -> b))
   , Arbitrary (m (b -> c)))
   => m (a, b, c) -> TestBatch
```

First, a trick for managing functions like this. We know it's going to want `Arbitrary` instances for the `Applicative` structure, functions (from `a` to `b` and `b` to `c`) embedded in that structure, and that it wants `EqProp` instances. That's all well and good, but we can ignore it:

```
    m (a, b, c) -> TestBatch
```

We just care about `m (a, b, c) -> TestBatch`. We could pass an actual value giving us our `Applicative` structure and three values that could be of different types but don't have to be. We could also pass a bottom with a type assigned to let it know what to randomly generate for validating the `Applicative` instance:

```
Prelude> xs = [("b", "w", 1)]
Prelude> quickBatch $ applicative xs

applicative:
  identity:     +++ OK, passed 500 tests.
  composition:  +++ OK, passed 500 tests.
  homomorphism: +++ OK, passed 500 tests.
  interchange:  +++ OK, passed 500 tests.
  functor:      +++ OK, passed 500 tests.
```

Note that it defaulted the `1 :: Num a => a` in order not to have an ambiguous type. We would have had to specify that outside of GHCi. In the following example, we'll use a bottom to fire the type class dispatch:

```
Prelude> type SSI = (String, String, Int)
Prelude> :{
*Main| let trigger :: [SSI]
*Main|     trigger = undefined
*Main| :}
Prelude> quickBatch (applicative trigger)

applicative:
  identity:     +++ OK, passed 500 tests.
  composition:  +++ OK, passed 500 tests.
  homomorphism: +++ OK, passed 500 tests.
  interchange:  +++ OK, passed 500 tests.
  functor:      +++ OK, passed 500 tests.
```

Again, it's not evaluating the value you pass it. That value is just to let it know what types to use.

17.8 ZipList Monoid

The default monoid of lists in the GHC `Prelude` is concatenation, but there is another way to monoidally combine lists. Whereas the default list `mappend` ends up doing the following:

```
[1, 2, 3] <> [4, 5, 6]

-- changes to

[1, 2, 3] ++ [4, 5, 6]

[1, 2, 3, 4, 5, 6]
```

The `ZipList` monoid combines the values of the two lists as parallel sequences using a monoid provided by the values themselves to get the job done:

```
[1, 2, 3] <> [4, 5, 6]

-- changes to
```

```
[
  1 <> 4
, 2 <> 5
, 3 <> 6
]
```

This should remind you of functions like `zip` and `zipWith`.

To make the above example work, you can assert a type like `Sum Integer` for the `Num` values to get a `Monoid`:

```
Prelude> import Data.Monoid
Prelude> 1 <> 2

• Ambiguous type variable ‘a0’ arising from
    a use of ‘print’
  prevents the constraint ‘(Show a0)’ from
    being solved.
  Probable fix: use a type annotation to
    specify what ‘a0’ should be.
... more noise about instances ...

Prelude> 1 <> (2 :: Sum Integer)
Sum {getSum = 3}
```

`Prelude` doesn't provide this `Monoid` for us, so we must define it ourselves:

```
module Apl1 where

import Control.Applicative
import Data.Monoid
import Test.QuickCheck
import Test.QuickCheck.Checkers
import Test.QuickCheck.Classes
```

Some unfortunate orphan instances follow. Try to avoid these in code you're going to keep or release:

```
-- this isn't going to work properly
instance Monoid a
      => Monoid (ZipList a) where
  mempty  = ZipList []
  mappend = liftA2 mappend

instance Arbitrary a
      => Arbitrary (ZipList a) where
  arbitrary = ZipList <$> arbitrary

  instance Arbitrary a
        => Arbitrary (Sum a) where
  arbitrary = Sum <$> arbitrary

instance Eq a
      => EqProp (ZipList a) where
  (=-=) = eq
```

If we fire this up in the REPL, and test for its validity as a `Monoid`, it'll fail:

```
Prelude> zl = ZipList [1 :: Sum Int]
Prelude> quickBatch $ monoid zl

monoid:
  left  identity:
   *** Failed! Falsifiable (after 3 tests):
ZipList [ Sum {getSum = -1} ]
  right identity:
   *** Failed! Falsifiable (after 4 tests):
ZipList [ Sum {getSum = -1}
        , Sum {getSum = 3}
        , Sum {getSum = 2} ]
  associativity:  +++ OK, passed 500 tests.
```

The problem is that the empty `ZipList` is the *zero* and not the *identity*!

Zero vs. Identity

```
-- Zero
n * 0 == 0

-- Identity
n * 1 == n
```

So how do we get an identity for `ZipList`?

```
Sum 1 `mappend` ??? -> Sum 1
```

```
instance Monoid a
      => Monoid (ZipList a) where
  mempty  = pure mempty
  mappend = liftA2 mappend
```

You'll find out what the `pure` does here when you write the `Applicative` for `ZipList` yourself.

List Applicative exercise

Implement `Applicative` for lists. Writing a minimally complete `Applicative` instance calls for writing the definitions of both `pure` and `<*>`. We're going to provide a hint, as well. Use the `checkers` library to validate your `Applicative` instance:

```
data List a =
    Nil
  | Cons a (List a)
  deriving (Eq, Show)
```

Remember what you wrote for the list `Functor`:

```
instance Functor List where
  fmap = undefined
```

Writing the list `Applicative` is similar:

```
instance Applicative List where
  pure = undefined
  (<*>) = undefined
```

Expected result:

```
Prelude> f = Cons (+1) (Cons (*2) Nil)
Prelude> v = Cons 1 (Cons 2 Nil)
Prelude> f <*> v
Cons 2 (Cons 3 (Cons 2 (Cons 4 Nil)))
```

In case you get stuck, use the following functions and hints:

```
append :: List a -> List a -> List a
append Nil ys = ys
append (Cons x xs) ys =
  Cons x $ xs `append` ys

fold :: (a -> b -> b) -> b -> List a -> b
fold _ b Nil      = b
fold f b (Cons h t) = f h (fold f b t)

concat' :: List (List a) -> List a
concat' = fold append Nil

-- write this one in terms
-- of concat' and fmap
flatMap :: (a -> List b)
        -> List a
        -> List b
flatMap f as = undefined
```

Use the above, and also try using `flatMap` and `fmap` without explicitly pattern matching on cons cells. You'll still need to handle the `Nil` cases.

`flatMap` is less strange than it would initially seem. It's basically "`fmap`, then smush":

```
Prelude> fmap (\x -> [x, 9]) [1, 2, 3]
[[1,9],[2,9],[3,9]]

Prelude> toMyList = foldr Cons Nil
Prelude> xs = toMyList [1, 2, 3]
Prelude> c = Cons
```

```
Prelude> f x = x `c` (9 `c` Nil)
Prelude> flatMap f xs
Cons 1 (Cons 9 (Cons 2
         (Cons 9 (Cons 3 (Cons 9 Nil)))))
```

`Applicative`, unlike `Functor`, is not guaranteed to have a unique implementation for a given datatype.

ZipList Applicative exercise

Implement the `ZipList` `Applicative`. Use the `checkers` library to validate your `Applicative` instance. We're going to provide the `EqProp` instance and explain the weirdness in a moment:

```
newtype ZipList' a =
  ZipList' [a]
  deriving (Eq, Show)

instance Eq a => EqProp (ZipList' a) where
  xs =-= ys = xs' `eq` ys'
    where xs' = let (ZipList' l) = xs
                in take 3000 l
          ys' = let (ZipList' l) = ys
                in take 3000 l

instance Functor ZipList' where
  fmap f (ZipList' xs) =
    ZipList' $ fmap f xs

instance Applicative ZipList' where
  pure = undefined
  (<*>) = undefined
```

The idea is to align a list of functions with a list of values and apply the first function to the first value and so on. The instance should also work with infinite lists. Some examples:

```
Prelude> zl' = ZipList'
Prelude> z = zl' [(+9), (*2), (+8)]
```

```
Prelude> z' = zl' [1..3]
Prelude> z <*> z'
ZipList' [10,4,11]
Prelude> z' = pure 1
Prelude> z <*> z'
ZipList' [10,2,9]
Prelude> z'' = zl' [1, 2]
Prelude> pure id <*> z''
ZipList' [1, 2]
```

Here, `toMyList` is whatever function you've written to convert from the built-in list type to your handmade `List` type. Note that the second `z'` is an infinite list. Check `Prelude` for functions that can give you what you need. One starts with the letter z, the other with the letter r. You're meant to search for these yourself. You're looking for inspiration from these functions, not to be able to directly reuse them, as you're using a custom `List` type, not the provided `Prelude` list type.

Explaining and justifying the weird EqProp The good news is it's `EqProp` that has the weird "check only the first 3,000 values" semantics instead of making the `Eq` instance weird. The bad news is this is a byproduct of testing for equality between infinite lists… that is, you can't. If you use a typical `EqProp` instance, the test for homomorphism in your `Applicative` instance will chase the infinite lists forever. Since `QuickCheck` is already an exercise in "good enough" validity checking, we could choose to feel justified in this. If you don't believe us, try running the following in your REPL:

```
repeat 1 == repeat 1
```

Either and Validation Applicative

Yep, here we go again with the types!

Specializing the types

```
-- f ~ Either e
```

```
type E = Either
(<*>) ::   f (a -> b) ->   f a ->   f b
(<*>) :: E e (a -> b) -> E e a -> E e b

pure :: a ->   f a
pure :: a -> E e a
```

Either vs. Validation

Often, the interesting part of an `Applicative` is the monoid. One byproduct of this is that just as you can have more than one valid `Monoid` for a given datatype, unike `Functor`, `Applicative` can have more than one valid and lawful instance for a given datatype.

The following is a brief demonstration of `Either`:

```
Prelude> pure 1 :: Either e Int
Right 1
Prelude> Right (+1) <*> Right 1
Right 2
Prelude> Right (+1) <*> Left ":("
Left ":("
Prelude> Left ":(" <*> Right 1
Left ":("
Prelude> Left ":(" <*> Left "sadface.png"
Left ":("
```

We've covered the benefits of `Either` already, and we've shown you what the `Maybe Applicative` can clean up, so we won't belabor those points. There's an alternative to `Either`, called `Validation`, that differs only in the `Applicative` instance:

```
data Validation err a =
    Failure err
  | Success a
  deriving (Eq, Show)
```

One thing to realize is that this is *identical* to the `Either` datatype, and there is even a pair of total functions that can go between `Validation` and `Either` values interchangeably. Remember when we mentioned natural transformations? Both of these functions are natural transformations:

```
validationToEither :: Validation e a
                   -> Either e a
validationToEither (Failure err) = Left err
validationToEither (Success a) = Right a

eitherToValidation :: Either e a
                   -> Validation e a
eitherToValidation (Left err) = Failure err
eitherToValidation (Right a) = Success a

eitherToValidation . validationToEither
  == id
validationToEither . eitherToValidation
  == id
```

How does `Validation` differ? Principally, in what the `Applicative` instance does with errors. Rather than just short-circuiting when it has two error values, it'll use the `Monoid` type class to combine them. Often, this will just be a list or set of errors, but you can do whatever you want:

```
data Errors =
   DividedByZero
 | StackOverflow
 | MooglesChewedWires
 deriving (Eq, Show)

success =   Success (+1)
        <*> Success 1

success == Success 2

failure =   Success (+1)
        <*> Failure [StackOverflow]

failure == Failure [StackOverflow]

failure' =   Failure [StackOverflow]
         <*> Success (+1)

failure' == Failure [StackOverflow]
```

```
failures =
      Failure [MooglesChewedWires]
  <*> Failure [StackOverflow]

failures ==
  Failure [MooglesChewedWires
         , StackOverflow]
```

With the value `failures`, we see what distinguishes `Either` and `Validation`: we can now preserve *all* failures that occur, not just the first one.

Exercise: Variations on Either

`Validation` has the same representation as `Either`, but it can be different. The `Functor` will behave the same, but the `Applicative` will be different. See above for an idea of how `Validation` should behave. Use the `checkers` library:

```
data Validation e a =
    Failure e
  | Success a
  deriving (Eq, Show)

-- same as Either
instance Functor (Validation e) where
  fmap = undefined

-- This is different
instance Monoid e =>
         Applicative (Validation e) where
  pure  = undefined
  (<*>) = undefined
```

17.9 Chapter exercises

Given a type that has an instance of `Applicative`, specialize the types of the methods. Test your specialization in the REPL. One way to do this is to bind aliases of the type class methods to more concrete types that have the type we tell you to fill in:

1. ```
   -- Type
   []

   -- Methods
   pure  :: a -> ? a
   (<*>) :: ? (a -> b) -> ? a -> ? b
   ```

2. ```
   -- Type
   IO

   -- Methods
   pure  :: a -> ? a
   (<*>) :: ? (a -> b) -> ? a -> ? b
   ```

3. ```
   -- Type
   (,) a

   -- Methods
   pure  :: a -> ? a
   (<*>) :: ? (a -> b) -> ? a -> ? b
   ```

4. ```
   -- Type
   (->) e

   -- Methods
   pure  :: a -> ? a
   (<*>) :: ? (a -> b) -> ? a -> ? b
   ```

Write instances for the following datatypes. Confused? Write out what the types should be. Use the `checkers` library to validate the instances:

1. ```
   data Pair a = Pair a a deriving Show
   ```

2. This should look familiar:

   ```
   data Two a b = Two a b
   ```

3. ```
   data Three a b c = Three a b c
   ```

4. `data Three' a b = Three' a b b`

5. `data Four a b c d = Four a b c d`

6. `data Four' a b = Four' a a a b`

Combinations

Remember the vowels and stops exercise from Chapter 10, on folds? Write a function to generate all the possible combinations of three input lists, using `liftA3` from `Control.Applicative`:

```
import Control.Applicative (liftA3)

stops :: String
stops = "pbtdkg"

vowels :: String
vowels = "aeiou"

combos :: [a] -> [b] -> [c] -> [(a, b, c)]
combos = undefined
```

17.10 Definitions

1. `Applicative` can be thought of as characterizing monoidal functors in Haskell. For a Haskeller's purposes, it's a way to functorially apply a function that is embedded in a structure `f` of the same type as the value you're mapping it over:

   ```
   fmap  ::   (a -> b) -> f a -> f b

   (<*>) :: f (a -> b) -> f a -> f b
   ```

17.11 Follow-up resources

1. Tony Morris and Nick Partridge. *Validation library*. `http://hackage.haskell.org/package/validation`

2. Conor McBride and Ross Paterson. *Applicative programming with effects*.
`http://staff.city.ac.uk/~ross/papers/Applicative.html`

3. Jeremy Gibbons and Bruno C. d. S. Oliveira. *The Essence of the Iterator Pattern*.
`https://www.cs.ox.ac.uk/jeremy.gibbons/publications/iterator.pdf`

4. Ross Paterson. *Constructing Applicative Functors*.
`http://staff.city.ac.uk/~ross/papers/Constructors.html`

5. Sam Lindley, Philip Wadler, and Jeremy Yallop. *Idioms are oblivious, arrows are meticulous, monads are promiscuous*. Note that *idiom* means "applicative functor" and is a useful search term for published work on this pattern.
`http://homepages.inf.ed.ac.uk/slindley/papers/idioms-arrows-monads.pdf`

Chapter 18

Monad

> There is nothing so practical as a good theory.
>
> Kurt Lewin

18.1 Monad

Finally, we come to one of the most talked about structures in Haskell: the monad. Monads are not, strictly speaking, necessary to Haskell. Although the current Haskell standard does use monads for constructing and transforming IO actions, older implementations of Haskell did not. Monads are powerful and fun, but they do not define Haskell. Rather, monads are defined in terms of Haskell.

Monads are applicative functors, but they have something special about them that makes them different from and more powerful than either `<*>` or `fmap` alone. In this chapter, we will:

- Define `Monad`, its operations, and its laws.
- Look at several examples of monads in practice.
- Write the `Monad` instances for various types.
- Address some misinformation about monads.

18.2 Sorry—a monad is not a burrito

Well, then what the heck is a monad?[1]

As we said above, a monad is an applicative functor with some unique features that make it a bit more powerful than either alone. A functor maps a function over some structure; an applicative maps a function that is contained in some structure over some other structure and then combines the two layers of structure like `mappend`. So you can think of monads as another way of applying functions over structure, with a couple of additional features. We'll get to those features in a moment. For now, let's check out the type class definition and core operations.

If you are using GHC 7.10 or newer, you'll see an `Applicative` constraint in the definition of `Monad`, as it should be:

[1]Section title with all due respect and gratitude to Mark Jason Dominus, whose blog post, "Monads are like burritos" is a classic of its genre: `http://blog.plover.com/prog/burritos.html`.

```
class Applicative m => Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  (>>) :: m a -> m b -> m b
  return :: a -> m a
```

We're going to explore this in some detail. Let's start with the type class constraint on `m`.

Applicative m

Older versions of GHC did not have `Applicative` as a superclass of `Monad`. Given that `Monad` is stronger than `Applicative`, and `Applicative` is stronger than `Functor`, you can derive `Applicative` and `Functor` in terms of `Monad`, just as you can derive `Functor` in terms of `Applicative`. What does this mean? It means you can write `fmap` using monadic operations, and it works:

```
fmap f xs = xs >>= return . f
```

Try it for yourself:

```
Prelude> fmap (+1) [1..3]
[2,3,4]

Prelude> [1..3] >>= return . (+1)
[2,3,4]
```

This happens to be a law, not a convenience. `Functor`, `Applicative`, and `Monad` instances over a given type should have the same core behavior.

We'll explore the relationship between these classes more completely in a bit, but as part of understanding the type class definition above, it's important to understand this chain of dependency:

```
Functor -> Applicative -> Monad
```

Whenever you've implemented an instance of `Monad` for a type, you *necessarily* have an `Applicative` and a `Functor`, as well.

Core operations

The `Monad` type class defines three core operations, although you only need to define `>>=` for a minimally complete `Monad` instance. Let's look at all three:

```
(>>=) :: m a -> (a -> m b) -> m b
(>>) :: m a -> m b -> m b
return :: a -> m a
```

We can dispense with the last of those, `return`: it's just the same as `pure`. All it does is take a value and return it inside your structure, whether that structure is a list or `Just` or `IO`. We talked about it a bit, and used it, back in Chapter 13, and we covered `pure` in the `Applicative` chapter, so there isn't much else to say about it.

The next operator, `>>`, doesn't have an official English-language name, but we like to call it Mr. Pointy. Some people do refer to it as the sequencing operator, which we must admit is more informative than Mr. Pointy. Mr. Pointy sequences two actions while discarding any resulting value of the first action. `Applicative` has a similar operator as well, although we didn't talk about it in that chapter. We will see examples of this operator in the upcoming section on `do` syntax.

Finally, the big `bind`! The `>>=` operator is called *bind* and is—or, at least, comprises—what makes `Monad` special.

The novel part of Monad

Conventionally, when we use monads, we use the bind function: `>>=`. Sometimes, we use it directly, sometimes indirectly via `do` syntax. The question we should ask ourselves is, what's unique to `Monad`—at least from the point of view of types?

We already saw that it's not `return`; that's another name for `pure` from `Applicative`.

We also noted (and will see more clearly soon) that it also isn't `>>`, which also has a counterpart in `Applicative`.

And it also isn't `>>=`, at least not in its entirety. The type of `>>=` is visibly similar to that of `fmap` and `<*>`, which makes sense since monads are applicative functors. For the sake of making this maximally similar, we're going to change the `m` of `Monad` to `f`:

```
fmap :: Functor f
     => (a -> b) -> f a -> f b
<*>  :: Applicative f
     => f (a -> b) -> f a -> f b
>>=  :: Monad f
     => f a -> (a -> f b) -> f b
```

OK, so bind is quite similar to `<*>` and `fmap` but with the first two arguments flipped. Still, the idea of mapping a function over a value while bypassing its surrounding structure is not unique to `Monad`.

We can demonstrate this point by using `fmap` on a function of type `(a -> m b)` to make it more like `>>=`, and it will work. Nothing will stop us. We will continue using the tilde to represent rough equivalence between two things:

```
-- If b ~ f b

fmap :: Functor f
     => (a -> f b) -> f a -> f (f b)
```

Let's demonstrate this idea with list as our structure:

```
Prelude> andOne x = [x, 1]
Prelude> andOne 10
[10,1]

Prelude> :t fmap andOne [4, 5, 6]
fmap andOne [4, 5, 6] :: Num t => [[t]]

Prelude> fmap andOne [4, 5, 6]
[[4,1],[5,1],[6,1]]
```

But, lo! We know from our types that we end up with something of `f (f b)`—that is, an extra layer of structure, and now we have a result of nested lists. What if we want `Num a => [a]` instead of nested lists? We want a single layer of `f` structure, but our mapped function has itself *generated more structure*! After mapping a function that generates *additional* monadic structure in its return type, we want a way to discard one layer of that structure.

So how do we accomplish that? Well, we saw how to do what we want with lists very early on in this book:

```
Prelude> concat $ fmap andOne [4, 5, 6]
[4,1,5,1,6,1]
```

The type of `concat`, fully generalized:

```
concat :: Foldable t => t [a] -> [a]
```

We can assert a less general type for our purposes here:

```
concat :: [[a]] -> [a]
```

`Monad`, in a sense, is a generalization of `concat`! The unique part of `Monad` is the following function:

```
import Control.Monad (join)

join :: Monad m => m (m a) -> m a

-- compare

concat ::          [[a]]   -> [a]
```

It's somewhat novel that we can inject more structure via our function application, where applicatives and `fmap`s have to leave the structure untouched. Allowing the function itself to alter the structure is something we've not seen in `Functor` and `Applicative`, and we'll explore the ramifications of that ability more, especially when we start talking about the `Maybe` monad. But we *can* inject more structure with a standard `fmap` if we wish, as we saw above. However, the ability to flatten those two layers of structure into one is what makes `Monad` special. And it's by putting that `join` function together with the mapping function that we get the bind function, also known as `>>=`.

So how do we get that bind?

The answer is the exercise Write `bind` in terms of `fmap` and `join`.

Fear is the mind-killer, friend. You can do it:

```
-- keep in mind this is >>= flipped
bind :: Monad m => (a -> m b) -> m a -> m b
bind = undefined
```

What Monad is not

Since `Monad` is somewhat abstract and a little slippery, many people talk about it from one or two perspectives that they feel most comfortable with. Quite often, they address what `Monad` is from the perspective of the `IO Monad`. `IO` does have a `Monad` instance, and it is a very common use of monads. However, understanding monads only through that instance leads to limited intuitions for what monads are and what they can do, and to a lesser extent, a wrong notion of what `IO` is all about.

A monad is not:

1. Impure. Monadic functions are pure functions. `IO` is an abstract datatype that allows for impure, or effectful, actions, and it has a `Monad` instance. But there's nothing impure about monads.

2. An embedded language for imperative programming. Simon Peyton Jones, one of the lead developers and researchers of Haskell and its implementation in GHC, has famously said, "Haskell is the world's finest imperative programming language," and he was talking about the way monads handle effectful programming. While monads are often used for sequencing actions in a way that looks like imperative programming, there are commutative monads that do not order actions. We'll see one a few chapters down the line when we talk about `Reader`.

3. A value. The type class describes a specific relationship between elements in a domain and defines some operations over them. When we refer to something as "a monad," it's the same as when we talk about "a monoid" or "a functor." None of those are values.

4. About strictness. The monadic operations of bind and `return` are non-strict. Some operations can be made strict within a specific instance. We'll talk more about this later in the book.

Using monads also doesn't require knowing math. Or category theory. It does not require mystical trips to the tops of mountains or starving oneself in a desert somewhere.

The `Monad` type class is generalized structure manipulation with some laws to make it sensible. Just like `Functor` and `Applicative`. We sort of hate to diminish the mystique, but that's all there is to it.

Monad also lifts!

The `Monad` class also includes a set of `lift` functions that are the same as the ones we already saw in `Applicative`. They don't do anything different, but they are still around, because some libraries used them before applicatives were discovered, so the `liftM` set of functions still exists to maintain compatibility. You may still see them sometimes. We'll take a short tour of these functions, comparing them directly to their applicative counterparts:

```
liftA :: Applicative f
      => (a -> b)  -> f a  -> f b
liftM :: Monad m
      => (a1 -> r) -> m a1 -> m r
```

As you may recall, this is `fmap` with a different type class constraint. If you'd like to see examples of how it works, we encourage you to write `fmap` functions in your REPL and take turns replacing the `fmap` with `liftA` or `liftM`.

But that's not all we have:

```
liftA2 :: Applicative f
       => (a -> b -> c)
       -> f a
       -> f b
       -> f c

liftM2 :: Monad m
       => (a1 -> a2 -> r)
       -> m a1
       -> m a2
       -> m r
```

Aside from the numbering, these appear the same. Let's try them:

```
Prelude> liftA2 (,) (Just 3) (Just 5)
Just (3,5)

Prelude> liftM2 (,) (Just 3) (Just 5)
Just (3,5)
```

You may remember way back in Chapter 9, we talked about a function called `zipWith`. `zipWith` is `liftA2` or `liftM2` specialized to lists:

```
Prelude> :t zipWith
zipWith :: (a -> b -> c)
        -> [a] -> [b] -> [c]
Prelude> zipWith (+) [3, 4] [5, 6]
[8,10]
Prelude> liftA2 (+) [3, 4] [5, 6]
[8,9,9,10]
```

Well, the types are the same, but the behavior differs. The differing behavior has to do with which list monoid is being used.

All right. Then we have the threes:

```
liftA3 :: Applicative f
       => (a -> b -> c -> d)
       -> f a -> f b
       -> f c -> f d

liftM3 :: Monad m
       => (a1 -> a2 -> a3 -> r)
       -> m a1 -> m a2
       -> m a3 -> m r
```

And, coincidentally, there is also a `zipWith3` function. Let's see what happens:

```
Prelude> :t zipWith3
zipWith3 :: (a -> b -> c -> d) ->
            [a] -> [b] -> [c] -> [d]
```

```
Prelude> liftM3 (,,) [1, 2] [3] [5, 6]
[(1,3,5),(1,3,6),(2,3,5),(2,3,6)]
Prelude> zipWith3 (,,) [1, 2] [3] [5, 6]
[(1,3,5)]
```

Again, using a different monoid gives us a different set of results.

We wanted to introduce these functions here, because they will come up in some later examples in the chapter, but they aren't especially pertinent to `Monad`, and we saw the gist of them in the previous chapter. So, let's turn our attention back to monads, shall we?

18.3 do syntax and monads

We introduced `do` syntax in Chapter 13, when we built a hangman game. We were using it within the context of `IO` as syntactic sugar that allowed us to easily sequence actions by feeding the result of one action as the input value to the next. While `do` syntax works with any monad—not just `IO`—it is most commonly seen when using `IO`. This section is going to talk about why `do` is sugar and demonstrate what the `join` of `Monad` can do for us. We will be using the `IO Monad` to demonstrate here, but later on we'll see some examples of `do` syntax without `IO`.

To begin, let's look at some correspondences:

```
(*>) :: Applicative f => f a -> f b -> f b


(>>) :: Monad m =>       m a -> m b -> m b
```

For our purposes, `*>` and `>>` are the same thing: sequencing functions, but with two different constraints. They should in all cases do the same thing:

```
Prelude> let h = "Hello, "
Prelude> let w = "World!"
Prelude> putStrLn h >> putStrLn w
Hello,
World!
Prelude> putStrLn h *> putStrLn w
```

```
Hello,
World!
```

Not observably different. Good enough for government work!

We can see what `do` syntax looks like after the compiler desugars it for us by manually transforming it ourselves:

```
import Control.Applicative ((*>))

sequencing :: IO ()
sequencing = do
  putStrLn "blah"
  putStrLn "another thing"

sequencing' :: IO ()
sequencing' =
  putStrLn "blah" >>
  putStrLn "another thing"

sequencing'' :: IO ()
sequencing'' =
  putStrLn "blah" *>
  putStrLn "another thing"
```

You should have had the same results for each of the above. We can do the same with the variable binding that `do` syntax includes:

```
binding :: IO ()
binding = do
  name <- getLine
  putStrLn name

binding' :: IO ()
binding' =
  getLine >>= putStrLn
```

Instead of naming the variable and passing that as an argument to the next function, we use `>>=`, which passes it directly.

When fmap alone isn't enough

Note that if you try to `fmap` the `putStrLn` function over `getLine`, it won't do anything. Try typing this into your REPL:

```
Prelude> putStrLn <$> getLine
```

You've used `getLine`, so when you hit enter, it should await your input. Type something in, hit enter again, and see what happens.

Whatever input you give it doesn't print, although it seems like it should, due to the `putStrLn` being mapped over the `getLine`. We evaluate the IO action that requests input but not the one that prints it. So, what's happening?

Well, let's start with the types. The type of what you're trying to do is this:

```
Prelude> :t putStrLn <$> getLine
putStrLn <$> getLine :: IO (IO ())
```

We're going to break it down a little bit so that we'll understand why this won't work. First, `getLine` performs I/O to get a `String`:

```
getLine :: IO String
```

And `putStrLn` takes a `String` argument, performs I/O, and returns nothing interesting—parents of children with an allowance can sympathize:

```
putStrLn :: String -> IO ()
```

What is the type of `fmap` as it concerns `putStrLn` and `getLine`?

```
-- The type we start with
<$> :: Functor f => (a -> b) -> f a -> f b

-- Our (a -> b) is putStrLn
           (a      -> b   )
putStrLn :: String -> IO ()
```

That `b` gets specialized to the type `IO ()`, which is going to jam another IO action *inside* of the I/O that `getLine` performs. Perhaps this looks familiar from our demonstration of what happens when you use `fmap` to map a function with type `(a -> m b)` instead of type `(a -> b)`—that is what's happening here. This is what is happening with our types:

```
f :: Functor f => f String -> f (IO ())
f x = putStrLn <$> x

g :: (String -> b) -> IO b
g x = x <$> getLine

putStrLn <$> getLine :: IO (IO ())
```

OK... so, which `IO` is which, and why does it ask for input but not print what we type in?

```
--   [1]  [2]  [3]
h :: IO  (IO   ())
h = putStrLn <$> getLine
```

1. This outermost `IO` structure represents the effects `getLine` must perform to get you a `String` that the user types in.
2. This inner `IO` structure represents the effects that would be performed *if* `putStrLn` were evaluated.
3. The unit here is the unit that `putStrLn` returns.

One of the strengths of Haskell is that we can refer to, compose, and map over effectful computations without performing them or bending over backwards to make that pattern work. For a simpler example of how we can wait to evaluate IO actions (or any computation, in general), consider the following:

```
Prelude> printOne = putStrLn "1"
Prelude> printTwo = putStrLn "2"
Prelude> twoActions = (printOne, printTwo)
Prelude> :t twoActions
twoActions :: (IO (), IO ())
```

With that tuple of two IO actions defined, we can now grab one and evaluate it:

```
Prelude> fst twoActions
1
Prelude> snd twoActions
2
Prelude> fst twoActions
1
```

Note that we are able to evaluate IO actions multiple times. This will be significant later.

Back to our conundrum of why we can't `fmap` the `putStrLn` over `getLine`. Perhaps you've already figured out what we need to do. We need to join those two `IO` layers together. To get what we want, we need the unique thing that `Monad` offers: `join`. Watch it work:

```
Prelude> import Control.Monad (join)
Prelude> join $ putStrLn <$> getLine
blah
blah
Prelude> :t join $ putStrLn <$> getLine
join $ putStrLn <$> getLine :: IO ()
```

What `join` does here is *merge* the effects of `getLine` and `putStrLn` into a single IO action. This merged IO action performs the effects in the order determined by the nesting of the IO actions. As it happens, the cleanest way to express ordering in a lambda calculus without bolting on something unpleasant is through the nesting of expressions, or lambdas.

That's right. We still haven't left the lambda calculus behind. Monadic sequencing and `do` syntax seem on the surface to be very far removed from that. But they aren't. As we said, monadic actions are still pure, and the sequencing operations we use here are ways of nesting lambdas. Now, `IO` is a bit different, as it does allow for side effects, but since those effects are constrained within the `IO` type, all the rest of it is still a pure lambda calculus.

Sometimes, it is valuable to suspend or otherwise not perform an IO action until some determination is made, so types like `IO (IO ())`

aren't necessarily invalid, but you should be aware of what's needed to make this example work.

Let's get back to desugaring `do` syntax with our now-enriched understanding of what monads do for us:

```
bindingAndSequencing :: IO ()
bindingAndSequencing = do
  putStrLn "name pls:"
  name <- getLine
  putStrLn ("y helo thar: " ++ name)

bindingAndSequencing' :: IO ()
bindingAndSequencing' =
  putStrLn "name pls:" >>
  getLine >>=
  \name ->
    putStrLn ("y helo thar: " ++ name)
```

As the nesting intensifies, you can see how `do` syntax can make things a bit cleaner and easier to read:

```
twoBinds :: IO ()
twoBinds = do
  putStrLn "name pls:"
  name <- getLine

  putStrLn "age pls:"
  age <- getLine

  putStrLn ("y helo thar: "
            ++ name ++ " who is: "
            ++ age ++ " years old.")

twoBinds' :: IO ()
twoBinds' =
  putStrLn "name pls:" >>
   getLine >>=

    \name ->
     putStrLn "age pls:" >>
      getLine >>=
```

```
\age ->
 putStrLn ("y helo thar: "
           ++ name ++ " who is: "
           ++ age ++ " years old.")
```

18.4 Examples of Monad use

All right, we've seen what is different about `Monad` and seen a small demonstration of what that does for us. What we need now is to see how monads work in code, with `Monad`s other than `IO`.

List

We've been starting off our examples of these type classes in use with list examples, because they can be quite easy to see and understand. We will keep this section brief, though, as we have more exciting things to show you.

Specializing the types

This process should be familiar to you by now:

```
(>>=) ::  Monad m
      => m  a  ->  (a ->  m  b) ->  m  b
(>>=) :: [ ] a  ->  (a -> [ ] b) -> [ ] b
```

More commonly:

```
(>>=) ::    [a] ->  (a -> [b])   -> [b]
```

Same thing as `pure`:

```
return :: Monad m => a ->  m  a
return ::            a -> [ ] a
return ::            a -> [a]
```

Excellent. It's like `fmap`, except the order of arguments is flipped, and we can now generate more list (or an empty list) inside of our mapped function. Let's take it for a spin.

Example of the list monad in use

Let's start with a function and identify how the parts fit with our monadic types:

```
twiceWhenEven :: [Integer] -> [Integer]
twiceWhenEven xs = do
  x <- xs
  if even x
    then [x*x, x*x]
    else [x*x]
```

The `x <- xs` line binds individual values out of the list input, like a list comprehension, giving us an `a`. The if-then-else expression is our `a -> m b`. It takes the individual `a` values that have been bound out of our `m a` and can generate more values, thereby increasing the size of the list.

The `m a` that is our first input will be the argument we pass to it below:

```
Prelude> twiceWhenEven [1..3]
[1,4,4,9]
```

Now try this:

```
twiceWhenEven :: [Integer] -> [Integer]
twiceWhenEven xs = do
  x <- xs
  if even x
    then [x*x, x*x]
    else []
```

And try giving it the same input as above (for easy comparison). Is the result what you expect? Keep playing around with this, forming hypotheses about what will happen and why and testing them in the REPL to develop an intuition for how monads are working on a simple example. The examples in the next sections are longer and more complex.

Maybe Monad

Now, we come to a more exciting demonstration of what we can do with our newfound power.

Specializing the types

It is the season for examining the types:

```
-- type M = Maybe
-- m ~ Maybe

(>>=) :: Monad m
    => m    a -> (a ->     m b) ->     m b
(>>=) ::
      Maybe a -> (a -> Maybe b) -> Maybe b

-- same as pure
return :: Monad m => a ->     m a
return ::            a -> Maybe a
```

There should have been nothing surprising there, so let's get to the meat of the matter.

Using the Maybe Monad

This example looks like the one from the `Applicative` chapter, but it's different. We encourage you to compare the two, although we've been explicit about what exactly is happening here. You developed some intutions above for `do` syntax and the list `Monad`; here, we'll be explicit about what's happening, and by the time we get to the `Either` demonstration below, it should be clear. Let's get started:

```
data Cow = Cow {
      name   :: String
    , age    :: Int
    , weight :: Int
   } deriving (Eq, Show)

noEmpty :: String -> Maybe String
noEmpty ""  = Nothing
noEmpty str = Just str
```

```
noNegative :: Int -> Maybe Int
noNegative n | n >= 0 = Just n
             | otherwise = Nothing

-- if Cow's name is Bess,
-- it must be under 500
weightCheck :: Cow -> Maybe Cow
weightCheck c =

  let w = weight c
      n = name c
  in if n == "Bess" && w > 499
     then Nothing
     else Just c

mkSphericalCow :: String
               -> Int
               -> Int
               -> Maybe Cow
mkSphericalCow name' age' weight' =

  case noEmpty name' of
   Nothing -> Nothing

   Just nammy ->
     case noNegative age' of
      Nothing -> Nothing

      Just agey ->
       case noNegative weight' of
        Nothing -> Nothing

        Just weighty ->
          weightCheck
            (Cow nammy agey weighty)
```

```
Prelude> mkSphericalCow "Bess" 5 499
Just (Cow {name = "Bess", age = 5,
  weight = 499})
Prelude> mkSphericalCow "Bess" 5 500
Nothing
```

First, we'll clean it up with `do` syntax, and then we'll see why we can't do this with `Applicative`:

```
mkSphericalCow' :: String
                -> Int
                -> Int
                -> Maybe Cow

mkSphericalCow' name' age' weight' = do
  nammy <- noEmpty name'
  agey <- noNegative age'
  weighty <- noNegative weight'
  weightCheck (Cow nammy agey weighty)
```

And this works as expected:

```
Prelude> mkSphericalCow' "Bess" 5 500
Nothing
Prelude> mkSphericalCow' "Bess" 5 499
Just (Cow {name = "Bess", age = 5,
  weight = 499})
```

Can we write it with `>>=`? Sure!

```
mkSphericalCow'' :: String
                 -> Int
                 -> Int
                 -> Maybe Cow
mkSphericalCow'' name' age' weight' =
  noEmpty name' >>=
  \nammy ->
    noNegative age' >>=
    \agey ->
     noNegative weight' >>=
     \weighty ->
      weightCheck (Cow nammy agey weighty)
```

So why can't we do this with `Applicative`? Because our `weightCheck` function depends on the prior existence of a `Cow` value and returns more monadic structure in its return type, `Maybe Cow`.

If your `do` syntax looks like this:

```
doSomething = do
  a <- f
  b <- g
  c <- h
  pure (a, b, c)
```

You can rewrite it using `Applicative`. On the other hand, if you have something like this:

```
doSomething' n = do
  a <- f n
  b <- g a
  c <- h b
  pure (a, b, c)
```

You're going to need `Monad`, because `g` and `h` are producing monadic structure based on values that can only be obtained by depending on values generated from monadic structure. You'll need `join` to crunch the nesting of monadic structure back down. If you don't believe us, try translating `doSomething'` to `Applicative`: so no resorting to `>>=` or `join`.

Here's some code to kick that around:

```
f :: Integer -> Maybe Integer
f 0 = Nothing
f n = Just n

g :: Integer -> Maybe Integer
g i =
  if even i
  then Just (i + 1)
  else Nothing

h :: Integer -> Maybe String
h i = Just ("10191" ++ show i)
```

```
doSomething' n = do
  a <- f n
  b <- g a
  c <- h b
  pure (a, b, c)
```

The long and short of it:

1. With the `Maybe Applicative`, each `Maybe` computation fails or succeeds independently of one another. You're lifting functions that are also `Just` or `Nothing` over `Maybe` values.
2. With the `Maybe Monad`, computations contributing to the final result can choose to return `Nothing` based on previous computations.

Exploding a spherical cow

We said we'd be quite explicit about what's happening in the above, so let's do this thing. Let's get into the guts of this code and how binding over `Maybe` values works.

For once, this example instance is what's in GHC's `base` library at the time of writing:

```
instance Monad Maybe where
  return x = Just x

  (Just x) >>= k      = k x
  Nothing  >>= _      = Nothing

mkSphericalCow'' :: String
                 -> Int
                 -> Int
                 -> Maybe Cow
```

```
mkSphericalCow'' name' age' weight' =
  noEmpty name' >>=
  \nammy ->
    noNegative age' >>=
    \agey ->
     noNegative weight' >>=
     \weighty ->
     weightCheck (Cow nammy agey weighty)
```

And what happens if we pass it some arguments?

```
mkSphericalCow'' "Bess" 5 499 =
  noEmpty "Bess" >>=
  \nammy ->
    noNegative 5 >>=
    \agey ->
     noNegative 499 >>=
     \weighty ->
     weightCheck (Cow nammy agey weighty)

-- "Bess" /= "", so skipping this pattern
-- noEmpty ""  = Nothing
noEmpty "Bess" = Just "Bess"
```

So, we produce the value `Just "Bess"`; however, `nammy` will be the `String` and not also the `Maybe` structure, because `>>=` passes `a` to the function it binds over the monadic value, not `m a`. Here, we'll use the `Maybe Monad` instance to examine why:

```
instance Monad Maybe where
  return x = Just x

  (Just x) >>= k      = k x
  Nothing  >>= _      = Nothing

  noEmpty "Bess" >>= \nammy ->
    (rest of the computation)
  -- noEmpty "Bess" evaluated
  -- to Just "Bess". So the first
  -- Just case matches.
```

```
(Just "Bess") >>= \nammy -> ...
(Just x) >>= k      = k x
-- k is \nammy et al.
-- x is "Bess" by itself.
```

So `nammy` is bound to `"Bess"`, and the following is the whole `k`:

```
\"Bess" ->
  noNegative 5 >>=
  \agey ->
   noNegative 499 >>=
   \weighty ->
   weightCheck (Cow nammy agey weighty)
```

Then, how does the age check go?

```
mkSphericalCow'' "Bess" 5 499 =
  noEmpty "Bess" >>=
  \"Bess" ->
    noNegative 5 >>=
    \agey ->
     noNegative 499 >>=
     \weighty ->
     weightCheck (Cow "Bess" agey weighty)

-- 5 >= 0 is true, so we get Just 5
noNegative 5 | 5 >= 0 = Just 5
             | otherwise = Nothing
```

Again, although `noNegative` returns `Just 5`, the bind function `>>=` will pass *5* on:

```
mkSphericalCow'' "Bess" 5 499 =
  noEmpty "Bess" >>=
  \"Bess" ->
    noNegative 5 >>=
    \5 ->
     noNegative 499 >>=
     \weighty ->
     weightCheck (Cow "Bess" 5 weighty)
```

```
-- 499 >= 0 is true, so we get  Just 499
noNegative 499 | 499 >= 0 = Just 499
               | otherwise = Nothing
```

Passing 499 on:

```
mkSphericalCow'' "Bess" 5 499 =
  noEmpty "Bess" >>=
  \"Bess" ->
    noNegative 5 >>=
    \5 ->
     noNegative 499 >>=
     \499 ->
     weightCheck (Cow "Bess" 5 499)

weightCheck (Cow "Bess" 5 499) =
  let 499 = weight (Cow "Bess" 5 499)
      "Bess" = name (Cow "Bess" 5 499)

  -- fyi, 499 > 499 is False.
  in if "Bess" == "Bess" && 499 > 499
     then Nothing
     else Just (Cow "Bess" 5 499)
```

So in the end, we return `Just (Cow "Bess" 5 499)`.

Fail fast, like an overfunded startup

But what if we had failed? We'll dissect the following computation:

```
Prelude> mkSphericalCow'' "" 5 499
Nothing
```

And how do the guts fall when we explode this poor bovine?

```
mkSphericalCow'' "" 5 499 =
  noEmpty "" >>=
  \nammy ->
    noNegative 5 >>=
    \agey ->
     noNegative 499 >>=
     \weighty ->
     weightCheck (Cow nammy agey weighty)

-- "" == "", so we get the Nothing case
noEmpty ""  = Nothing
-- noEmpty str = Just str
```

After we've evaluated `noEmpty ""` and gotten a `Nothing` value, we use `>>=`. How does that go?

```
instance Monad Maybe where
  return x = Just x

  (Just x) >>= k      = k x
  Nothing  >>= _      = Nothing

  -- noEmpty "" := Nothing
  Nothing >>=
    \nammy ->

  -- Just case doesn't match, so skip it.
  -- (Just x) >>= k      = k x

  -- This is what we're doing.
  Nothing  >>= _      = Nothing
```

So it turns out that `>>=` will drop the entire rest of the computation on the floor the moment *any* of the functions participating in the `Maybe Monad` actions produce a `Nothing` value:

```
mkSphericalCow'' "" 5 499 =
  Nothing >>= -- NOPE.
```

In fact, you can demonstrate to yourself that that stuff never gets used with `bottom` but does with a `Just` value:

```
Prelude> Nothing >>= undefined
Nothing
Prelude> Just 1 >>= undefined
*** Exception: Prelude.undefined
```

But why do we use the `Maybe` `Applicative` and `Monad`? Because this:

```
mkSphericalCow' :: String
                -> Int
                -> Int
                -> Maybe Cow

mkSphericalCow' name' age' weight' = do
  nammy <- noEmpty name'
  agey <- noNegative age'
  weighty <- noNegative weight'
  weightCheck (Cow nammy agey weighty)
```

Is a lot nicer than case matching the `Nothing` case over and over just so we can say `Nothing -> Nothing` a million times. Life is too short for repetition when computers *love* taking care of repetition.

Either

Whew. Let's all be thankful that cow was full of `Maybe` values and not tripe. Moving along, we're going to demonstrate the use of the `Either` `Monad`, step back a bit, and let your intuitions and what you have learned about `Maybe` guide you through.

Specializing the types

As always, we present the types:

```
-- m ~ Either e
(>>=) :: Monad m
      =>        m a
      -> (a ->        m b)
      ->        m b
(>>=) :: Either e a
      -> (a -> Either e b)
      -> Either e b
```

```
-- same as pure
return :: Monad m => a ->        m aq
return ::            a -> Either e a
```

Why do we keep doing this? To remind you that the types always show you the way, once you've figured them out.

Using the Either Monad

Use what you know to go carefully through this code and follow the types. First, we define our datatypes:

```
module EitherMonad where

-- years ago
type Founded = Int
-- number of programmers
type Coders = Int

data SoftwareShop =
  Shop {
      founded     :: Founded
    , programmers :: Coders
  } deriving (Eq, Show)

data FoundedError =
    NegativeYears Founded
  | TooManyYears Founded
  | NegativeCoders Coders
  | TooManyCoders Coders
  | TooManyCodersForYears Founded Coders
  deriving (Eq, Show)
```

Let's bring in some functions now:

```
validateFounded
  :: Int
  -> Either FoundedError Founded
```

```
validateFounded n
  | n < 0     = Left $ NegativeYears n
  | n > 500   = Left $ TooManyYears n
  | otherwise = Right n

-- Though, many programmers *are* negative.
validateCoders
  :: Int
  -> Either FoundedError Coders

validateCoders n
  | n < 0     = Left $ NegativeCoders n
  | n > 5000  = Left $ TooManyCoders n
  | otherwise = Right n

mkSoftware
  :: Int
  -> Int
  -> Either FoundedError SoftwareShop

mkSoftware years coders = do
  founded     <- validateFounded years
  programmers <- validateCoders coders

  if programmers > div founded 10
    then Left $
           TooManyCodersForYears
           founded programmers
    else Right $ Shop founded programmers
```

Note that `Either` always short-circuits on the *first* thing to have failed. It *must* because in the `Monad`, later values can depend on previous ones:

```
Prelude> mkSoftware 0 0
Right (Shop {founded = 0, programmers = 0})

Prelude> mkSoftware (-1) 0
Left (NegativeYears (-1))
```

```
Prelude> mkSoftware (-1) (-1)
Left (NegativeYears (-1))

Prelude> mkSoftware 0 (-1)
Left (NegativeCoders (-1))

Prelude> mkSoftware 500 0
Right (Shop {founded = 500,
  programmers = 0})

Prelude> mkSoftware 501 0
Left (TooManyYears 501)

Prelude> mkSoftware 501 501
Left (TooManyYears 501)

Prelude> mkSoftware 100 5001
Left (TooManyCoders 5001)

Prelude> mkSoftware 0 500
Left (TooManyCodersForYears 0 500)
```

So, there is no `Monad` for `Validation`. `Applicative` and `Monad` instances must have the same behavior. This is usually expressed in the form:

```
import Control.Monad (ap)

(<*>) == ap
```

This is a way of saying the `Applicative` apply for a type must not change behavior if derived from the `Monad` instance's bind operation:

```
-- Keeping in mind
(<*>) :: Applicative f
      => f (a -> b) -> f a -> f b
ap :: Monad m
      => m (a -> b) -> m a -> m b
```

Then deriving `Applicative` `<*>` from the stronger instance:

```
ap :: (Monad m) => m (a -> b) -> m a -> m b
ap m m' = do
  x  <- m
  x' <- m'
  return (x x')
```

The problem is that you can't make a `Monad` for `Validation` that accumulates the errors like the `Applicative` does. Instead, any `Monad` instance for `Validation` would be identical to the `Monad` instance of `Either`.

Short Exercise: Either Monad

Implement the `Either` `Monad`:

```
data Sum a b =
    First a
  | Second b
  deriving (Eq, Show)

instance Functor (Sum a) where
  fmap = undefined

instance Applicative (Sum a) where
  pure = undefined
  (<*>) = undefined

instance Monad (Sum a) where
  return = pure
  (>>=) = undefined
```

18.5 Monad laws

The `Monad` type class has laws, as the other type classes do. These laws exist, as with all the other type class laws, to ensure that your code does nothing surprising or harmful. If the `Monad` instance you write for your type abides by these laws, then your monads should work as you want them to. To write your own instance, you only have to define a `>>=` operation, but you want your binding to be as predictable as possible.

Identity laws

`Monad` has two identity laws:

```
-- right identity
m >>= return     =  m

-- left identity
return x >>= f   =  f x
```

Basically, both of these laws are saying that `return` should be neutral and not perform any computation. We'll line them up with the type of `>>=` to clarify what's happening:

```
(>>=) :: Monad m
      => m a -> (a -> m b) -> m b
--       [1]       [2]        [3]
```

First, right identity:

```
return :: a -> m a

   m   >>= return     =  m
-- [1]       [2]         [3]
```

The `m` does represent an `m a` and an `m b`, respectively, so the structure is there even if it's not apparent from the way the law is written.

And left identity:

```
-- applying return to x gives us an
-- m a value to start

  return x >>= f   =  f  x
--     [1]    [2]      [3]
```

Like `pure`, `return` shouldn't change any of the behavior of the rest of the function. It is only there to put things into structure when we need to, and the existence of the structure should not affect the computation.

Associativity

The law of associativity is not so different from other laws of associativity we have seen. It does look a bit different because of the nature of `>>=`, however:

```
(m >>= f) >>= g  =  m >>= (\x -> f x >>= g)
```

Regrouping the functions should not have any impact on the final result, same as the associativity of `Monoid`. The syntax here, in which, for the right side of the equals sign, we have to pass in an `x` argument might seem confusing at first. So, let's look at it more carefully.

This side looks the way we expect it to:

```
(m >>= f) >>= g
```

But remember that `>>=` allows the result value of one function to be passed as input to the next, like function application but with our value at the left and successive functions proceeding to the right. Remember this code?

```
getLine >>= putStrLn
```

The IO action for `getLine` is evaluated first, then `putStrLn` is passed the input string that results from running `getLine`'s effects. This left-to-right is partly down to the history of `IO` in Haskell—it's so the "order" of the code reads top to bottom. We'll explain this more later in the book.

When we reassociate them, we need to apply `f` so that `g` has an input value of type `m a` to start the whole thing off. So, we pass in the argument `x` via an anonymous function:

```
m >>= (\x -> f x >>= g)
```

And bada bing, now nothing can slow this roll.

We're doing that thing again

Out of mercy, we'll be using `checkers` (not Nixon's dog) again. The argument the `Monad` `TestBatch` wants is identical to the `Applicative`, a tuple of three value types embedded in the structural type:

```
Prelude> quickBatch (monad [(1, 2, 3)])

monad laws:
  left  identity: +++ OK, passed 500 tests.
  right identity: +++ OK, passed 500 tests.
  associativity:  +++ OK, passed 500 tests.
```

Going forward, we'll be using this to validate `Monad` instances. Let's write a bad `Monad` to see what it can catch for us.

Bad monads and their denizens

We're going to write an invalid `Monad` (and `Functor`). You could pretend it's `Identity` with an integer thrown in that gets incremented on each `fmap` or bind:

```
module BadMonad where

import Test.QuickCheck
import Test.QuickCheck.Checkers
import Test.QuickCheck.Classes

data CountMe a =
  CountMe Integer a
  deriving (Eq, Show)

instance Functor CountMe where
  fmap f (CountMe i a) =
    CountMe (i + 1) (f a)

instance Applicative CountMe where
  pure = CountMe 0
  CountMe n f <*> CountMe n' a =
    CountMe (n + n') (f a)

instance Monad CountMe where
  return = pure

  CountMe n a >>= f =
    let CountMe _ b = f a
    in CountMe (n + 1) b
```

```
instance Arbitrary a
      => Arbitrary (CountMe a) where
  arbitrary =
    CountMe <$> arbitrary <*> arbitrary

instance Eq a => EqProp (CountMe a) where
  (=-=) = eq

main = do
  let trigger :: CountMe (Int, String, Int)
      trigger = undefined
  quickBatch $ functor trigger
  quickBatch $ applicative trigger
  quickBatch $ monad trigger
```

When we run the tests, the `Functor` and `Monad` will fail top to bottom. The `Applicative` technically only fails the laws because `Functor` does. In the `Applicative` instance, we use a proper monoid-of-structure:

```
Prelude> main

functor:
  identity: *** Failed! Falsifiable
  (after 1 test):
CountMe 0 0
  compose:  *** Failed! Falsifiable
  (after 1 test):
<function>
<function>
CountMe 0 0

applicative:
  identity:     +++ OK, passed 500 tests.
  composition:  +++ OK, passed 500 tests.
  homomorphism: +++ OK, passed 500 tests.
  interchange:  +++ OK, passed 500 tests.
  functor:      *** Failed! Falsifiable
  (after 1 test):
<function>
```

```
CountMe 0 0

monad laws:
  left  identity: *** Failed! Falsifiable
  (after 1 test):
<function>
0
  right identity: *** Failed! Falsifiable
  (after 1 test):
CountMe 0 0
  associativity:  *** Failed! Falsifiable
  (after 1 test):
CountMe 0 0
```

We can reapply the weird, broken increment semantics and get a broken `Applicative`, as well:

```
instance Applicative CountMe where
  pure = CountMe 0

  CountMe n f <*> CountMe _ a =
    CountMe (n + 1) (f a)
```

Now, it's *all* broken:

```
applicative:
  identity:
    *** Failed! Falsifiable (after 1 test):
CountMe 0 0
  composition:
    *** Failed! Falsifiable (after 1 test):
CountMe 0 <function>
CountMe 0 <function>
CountMe 0 0
  homomorphism:
    *** Failed! Falsifiable (after 1 test):
<function>
0
  interchange:
    *** Failed! Falsifiable
```

```
  (after 3 tests):
CountMe (-1) <function>
0
```

Understanding what makes sense structurally for a `Functor`, `Applicative`, and `Monoid` can tell you what is potentially an invalid `Monad` instance before you've written any code. Incidentally, even if you fix the `Functor` and `Applicative` instances, the `Monad` instance is not yet fixed:

```
instance Functor CountMe where
  fmap f (CountMe i a) = CountMe i (f a)

instance Applicative CountMe where
  pure = CountMe 0
  CountMe n f <*> CountMe n' a =
    CountMe (n + n') (f a)

instance Monad CountMe where
  return = pure

  CountMe _ a >>= f = f a
```

This'll pass as a valid `Functor` and `Applicative`, but it's not a valid `Monad`. The problem is that while using `pure` to set the integer value to zero is fine for the purposes of the `Applicative`, it violates the right identity law of `Monad`:

```
Prelude> CountMe 2 "blah" >>= return
CountMe 0 "blah"
```

Our `pure` is too opinionated. It's still a valid `Applicative` and `Functor`, but what if `pure` doesn't agree with the `Monoid` of the structure? The following will pass the `Functor` laws, but it isn't a valid `Applicative`:

```
instance Functor CountMe where
  fmap f (CountMe i a) = CountMe i (f a)

instance Applicative CountMe where
  pure = CountMe 1
  CountMe n f <*> CountMe n' a =
    CountMe (n + n') (f a)
```

As it happens, if we change the monoid-of-structure to match the identity such that we have addition and the number zero, it's a valid `Applicative` again:

```
instance Applicative CountMe where
  pure = CountMe 0
  CountMe n f <*> CountMe n' a =
    CountMe (n + n') (f a)
```

As you gain experience with these structures, you'll learn to identify what might have a valid `Applicative` but no valid `Monad` instance. But how do we fix the `Monad` instance? By fixing the underlying `Monoid`!

```
instance Monad CountMe where
  return = pure

  CountMe n a >>= f =
    let CountMe n' b = f a
    in CountMe (n + n') b
```

Once our `Monad` instance starts summing the counts like the `Applicative` does, it works fine! It can be easy at times to accidentally write an invalid `Monad` that type checks, so it's important to use `QuickCheck` to validate your `Monoid`, `Functor`, `Applicative`, and `Monad` instances.

18.6 Application and composition

What we've seen so far has been primarily about function application. We probably weren't thinking too much about the relationship between function application and composition, because with `Functor` and `Applicative`, it doesn't matter much. Both concern functions that look like the usual `(a -> b)` arrangement, so composition "just works," and that this is true is guaranteed by the laws of those type classes:

```
fmap id = id

-- guarantees

fmap f . fmap g = fmap (f . g)
```

Which means composition under functors just works:

```
Prelude> fmap ((+1) . (+2)) [1..5]
[4,5,6,7,8]
Prelude> fmap (+1) . fmap (+2) $ [1..5]
[4,5,6,7,8]
```

With `Monad`, the situation seems less neat at first. Let's attempt to define composition for monadic functions in a simple way:

```
mcomp :: Monad m =>
         (b -> m c)
      -> (a -> m b)
      ->  a -> m c
mcomp f g a = f (g a)
```

If we try to load this, we'll get an error like this:

```
• Occurs check: cannot construct the
    infinite type: b ~ m b
• In the first argument of ‘f’,
    namely ‘(g a)’
  In the expression: f (g a)
  In an equation for ‘mcomp’:
    mcomp f g a = f (g a)
• Relevant bindings include
    g :: a -> m b
    f :: b -> m c
    mcomp :: (b -> m c) -> (a -> m b) ->
      a -> m c
```

Well, that doesn't work. That error message is telling us that `f` is expecting a `b` for its first argument, but `g` is passing an `m b` to `f`. So, how do we apply a function in the presence of some context that we want to ignore? We use `fmap`. That's going to give us an `m (m c)` instead of an `m c`, so we'll want to `join` those two monadic structures:

```
mcomp :: Monad m =>
         (b -> m c)
      -> (a -> m b)
       -> a -> m c
mcomp f g a = join (f <$> (g a))
```

But using `join` and `fmap` together means we can just use `>>=`:

```
mcomp'' :: Monad m =>
           (b -> m c)
        -> (a -> m b)
         -> a -> m c
mcomp'' f g a = g a >>= f
```

You don't need to write anything special to make monadic functions compose (as long as the monadic contexts are the same `Monad`), because Haskell has it covered: what you want is *Kleisli composition*. Don't sweat the strange name; it's not as weird as it sounds. As we saw above, what we need is function composition written in terms of `>>=` to allow us to deal with the extra structure, and that's what the Kleisli fish gives us.

Let's remind ourselves of the types of ordinary function composition and `>>=`:

```
(.) :: (b -> c) -> (a -> b) -> a -> c
```

```
(>>=) :: Monad m
      => m a -> (a -> m b) -> m b
```

To get Kleisli composition off the ground, we have to flip some arguments around to make the types work:

```
import Control.Monad

-- the order is flipped to match >>=

(>=>)
  :: Monad m
  => (a -> m b) -> (b -> m c) -> a -> m c
```

See any similarities to something you know yet?

```
(>=>)
  :: Monad m
  => (a -> m b) -> (b -> m c) -> a -> m c


flip (.)
  :: (a -> b)   -> (b -> c)   -> a -> c
```

It's function composition with monadic structure hanging off the functions we're composing. Let's see an example!

```
import Control.Monad ((>=>))

sayHi :: String -> IO String
sayHi greeting = do
  putStrLn greeting
  getLine

readM :: Read a => String -> IO a
readM = return . read

getAge :: String -> IO Int
getAge = sayHi >=> readM

askForAge :: IO Int
askForAge =
  getAge "Hello! How old are you? "
```

We use `return` composed with `read` to turn it into something that provides monadic structure after being bound over the output of `sayHi`. We need the Kleisli composition operator to stitch `sayHi` and `readM` together:

```
sayHi :: String -> IO String
readM :: Read a => String -> IO a

--  [1]  [2] [3]
    (a -> m   b)
String -> IO  String
```

```
-- [4]      [5] [6]
-> (b     -> m   c)
   String -> IO  a

-- [7]     [8] [9]
->  a ->    m   c
   String  IO   a
```

1. The first type is the type of the input to `sayHi`, `String`.
2. The `IO` that `sayHi` performs in order to present a greeting and receive input.
3. The `String` input from the user that `sayHi` returns.
4. The `String` that `readM` expects as an argument and that `sayHi` will produce.
5. The `IO readM` returns into. Note that `return/pure` produce `IO` values that perform no I/O.
6. The `Int` that `readM` returns.
7. The original, initial `String` input that `sayHi` expects so it knows how to greet the user and ask for their age.
8. The final combined IO action that performs all effects necessary to produce the final result.
9. The value inside of the final IO action; in this case, this is the `Int` value that `readM` returns.

18.7 Chapter exercises

Write `Monad` instances for the following types. Use the `QuickCheck` properties we showed you to validate your instances.

1. Welcome to the `Nope Monad`, where nothing happens and nobody cares:

```
data Nope a =
  NopeDotJpg

-- We're serious. Write it anyway.
```

2. ```
   data BahEither b a =
       PLeft a
     | PRight b
   ```

3. Write a `Monad` instance for `Identity`:

   ```
   newtype Identity a = Identity a
       deriving (Eq, Ord, Show)

   instance Functor Identity where
     fmap = undefined

   instance Applicative Identity where
     pure = undefined
     (<*>) = undefined

   instance Monad Identity where
     return = pure
     (>>=) = undefined
   ```

4. This one should be easier than the `Applicative` instance was. Remember to use the `Functor` that `Monad` requires, then see where the chips fall:

   ```
   data List a =
       Nil
     | Cons a (List a)
   ```

Write the following functions using the methods provided by `Monad` and `Functor`. Using stuff like identity and composition is fine, but it has to type check with the types provided:

1. ```
   j :: Monad m => m (m a) -> m a
   ```

   Expecting the following behavior:

   ```
   Prelude> j [[1, 2], [], [3]]
   [1,2,3]
   Prelude> j (Just (Just 1))
   Just 1
   Prelude> j (Just Nothing)
   ```

```
Nothing
Prelude> j Nothing
Nothing
```

2. `l1 :: Monad m => (a -> b) -> m a -> m b`

3. `l2 :: Monad m`
   `=> (a -> b -> c) -> m a -> m b -> m c`

4. `a :: Monad m => m a -> m (a -> b) -> m b`

5. You'll need recursion for this one:

   ```
   meh :: Monad m
       => [a] -> (a -> m b) -> m [b]
   ```

6. Hint: reuse `meh`:

   ```
   flipType :: (Monad m) => [m a] -> m [a]
   ```

18.8 Definitions

1. *Monad* is a type class reifying an abstraction that is commonly used in Haskell. Instead of an ordinary function of type `a` to `b`, you're functorially applying a function that produces more structure itself and using `join` to reduce the nested structure that results:

   ```
   fmap ::    (a -> b)   -> f a -> f b

   (<*>) :: f (a -> b)   -> f a -> f b

   (=<<) ::   (a -> f b) -> f a -> f b
   ```

2. A *monadic function* is one that generates more structure after having already been lifted over monadic structure. Contrast the function arguments to `fmap` and `>>=`:

   ```
   fmap :: (a -> b) -> f a -> f b

   (>>=) :: m a -> (a -> m b) -> m b
   ```

The significant difference is that the result of `>>=` is `m b` and requires `join`ing the result after lifting the function over `m`. What does this mean? That depends on the `Monad` instance.

The distinction can be seen with ordinary function composition and Kleisli composition, as well:

```
(.)
  :: (b ->   c) -> (a ->   b) -> a ->   c


(>=>)
  :: Monad m
  => (a -> m b) -> (b -> m c) -> a -> m c
```

3. *Bind* is, unfortunately, a somewhat overloaded term. You first saw it used early in the book with respect to binding variables to values, such as with the following:

```
let x = 2 in x + 2
```

Where `x` is a variable bound to 2. However, when we're talking about a `Monad` instance, typically bind will refer to having used `>>=` to lift a monadic function over the structure. The distinction being:

```
-- lifting (a -> b) over f in f a
fmap :: (a -> b) -> f a -> f b


-- binding (a -> m b) over m in m a
(>>=) :: m a -> (a -> m b) -> m b
```

You'll sometimes see us talk about the use of the bind `do`-notation `<-` or `>>=` as "binding over." When we do, we mean that we're lifting a monadic function, and we'll eventually `join` or smush the structure back down when we're done monkeying around in the `Monad`. *Don't panic* if we're a little casual about describing the use of `<-` as binding over some `a` or binding `a` out of `m a`.

18.9 Follow-up resources

1. Haskell Wiki. *What a Monad is not*.
   `https://wiki.haskell.org/What_a_Monad_is_not`
2. Haskell Wikibook. *Understanding monads*.
3. Gabriel Gonzalez. *How to desugar Haskell code*.
4. Stephen Diehl. *What I Wish I Knew When Learning Haskell*.
   Section on Monads.
   `http://dev.stephendiehl.com/hask/#monads`
5. Stephen Diehl. *Monads Made Difficult*.
   `http://www.stephendiehl.com/posts/monads.html`
6. Brent Yorgey. *Typeclassopedia*
   `https://wiki.haskell.org/Typeclassopedia`

Chapter 19

Applying Structure

> I often repeat repeat myself, I often repeat repeat. I don't don't know why know why, I simply know that I I I am am inclined to say to say a lot a lot this way this way— I often repeat repeat myself, I often repeat repeat.
>
> Jack Prelutsky

19.1 Applied structure

We thought you'd like to see `Monoid`, `Functor`, `Applicative`, and `Monad` *in the wild*, as it were. Since we'd like to finish this book before we have grandchildren, this will *not* be accompanied by the painstaking explanations and exercise regime you've experienced up to this point. Don't understand something? Figure it out! We'll do our best to leave a trail of breadcrumbs, so you can follow up on the code we show you. Consider this a breezy survey of how Haskellers write code when they think no one is looking and a pleasant break from your regularly scheduled exercises. The code demonstrated will not always include all the necessary context to make it run, so don't expect to be able to load the snippets in GHCi and have them work.

If you don't have a lot of previous programming experience and some of the applications are difficult for you to follow, you might prefer to return to this chapter at a later time, once you start trying to read and use Haskell libraries for practical projects.

19.2 Monoid

Monoids are everywhere once you recognize the pattern and start looking for them, but we've tried to choose a few good examples to illustrate typical use cases.

Templating content in Scotty

Here, the `scotty` web framework's "Hello, World" example uses `mconcat` to inject the parameter `"word"` into the HTML page returned:

```
{-# LANGUAGE OverloadedStrings #-}
import Web.Scotty

import Data.Monoid (mconcat)

main = scotty 3000 $ do
    get "/:word" $ do
        beam <- param "word"
        html
        (mconcat
          [ "<h1>Scotty, "
          , beam
          , " me up!</h1>"])
```

If you're interested in following up on this example, you can find it and a tutorial on the `scotty` Github repository.

Concatenating connection parameters

The next example is from Aditya Bhargava's "Making A Website With Haskell," a blog post that walks you through several steps for, well, making a simple website in Haskell. It also uses the `scotty` web framework.

In this case, we're using `foldr` and `Monoid` to concatenate connection parameters for connecting to a database:

```
runDb :: SqlPersist (ResourceT IO) a
      -> IO a
runDb query = do
    let connStr =
          foldr (\(k,v) t ->
                t <> (encodeUtf8 $
                k <> "=" <> v <> " "))
          "" params
    runResourceT
      . withPostgresqlConn connStr
      $ runSqlConn query
```

If you're interested in following up on it, this blog post is one of many that shows you step by step how to use `scotty`, although many of them breeze through each step without a great deal of explanation. It will be easier to understand `scotty` in detail once you've worked through monad transformers, but if you'd like to start playing around with some basic projects, you may want to try them out.

Concatenating key configurations

The next example is going to be a bit meatier than the two previous ones.

`xmonad` is a windowing system for X11 written in Haskell. The configuration language is Haskell—the binary that runs your WM is compiled from your personal configuration. The following is an example of using `mappend` to combine the default configuration's key mappings and a modification of those keys:

```
import XMonad
import XMonad.Actions.Volume
import Data.Map.Lazy (fromList)
import Data.Monoid (mappend)

main = do
  xmonad def { keys =
    \c -> fromList [
      ((0, xK_F6),
       lowerVolume 4 >> return ()),
      ((0, xK_F7),
       raiseVolume 4 >> return ())
    ] `mappend` keys defaultConfig c
  }
```

The type of `keys` is a function:

```
keys :: !(XConfig Layout
       -> Map (ButtonMask, KeySym) (X ()))
```

You don't need to get too excited about the exclamation point right now—it's the syntax for a nifty thing called a *strictness annotation*, which makes a field in a product strict. That is, you won't be able

to construct the record or product that contains the value without also forcing that field to evaluate to weak head normal form. We'll explain this in more detail later in the book.

The gist of the `main` function above is that it allows your keymapping to be based on the current configuration of your environment. Whenever you type a key, `xmonad` will pass the current config to your `keys` function in order to determine what (if any) action it should take. We're using the `Monoid` here to add new keyboard shortcuts for lowering and raising the volume with F6 and F7. The monoid of the `keys` functions is combining all of the key maps each function produces when applied to the `XConfig` to produce a final canonical key map.

Say what?

This is a `Monoid` instance we didn't cover in the `Monoid` chapter, so let's take a look at it now:

```
instance Monoid b
      => Monoid (a -> b)
      -- Defined in GHC.Base
```

This, friends, is the `Monoid` of functions.

But how does it work? First, let's set up some very trivial functions for demonstration purposes:

```
Prelude> import Data.Monoid
Prelude> f = const (Sum 1)
Prelude> g = const (Sum 2)
Prelude> f 9001
Sum {getSum = 1}
Prelude> g 9001
Sum {getSum = 2}
```

Query the types of those functions, and see how you think they will match up to the `Monoid` instance above.

We know that whatever arguments we give to `f` and `g`, they will always return their first arguments, which are `Sum` monoids. So if we `mappend` `f` and `g`, they're going to ignore whatever argument we try to apply them to and use the `Monoid` to combine the results:

```
Prelude> (f <> g) 9001
Sum {getSum = 3}
```

So, this `Monoid` instance allows us to `mappend` the results of two function applications:

```
(a -> b) <> (a -> b)
```

Just as long as the `b` has a `Monoid` instance.

We're going to offer a few more examples that will get you closer to what the particular use of `mappend` in the `xmonad` example is doing. We mentioned `Data.Map` back in Chapter 14, on testing. It gives us ordered pairs of keys and values:

```
Prelude> import qualified Data.Map as M
Prelude M> :t M.fromList
M.fromList :: Ord k => [(k, a)] -> Map k a

Prelude M> let f = M.fromList [('a', 1)]
Prelude M> let g = M.fromList [('b', 2)]
Prelude M> :t f
f :: Num a => Map Char a
Prelude M> import Data.Monoid
Prelude M Data.Monoid> f <> g
fromList [('a',1),('b',2)]
Prelude M Data.Monoid> :t (f <> g)
(f <> g) :: Num a => Map Char a

Prelude M Data.Monoid> mappend f g
fromList [('a',1),('b',2)]
Prelude M Data.Monoid> f `mappend` g
fromList [('a',1),('b',2)]

-- but note what happens here:
Prelude> f <> g
fromList [('a',1)]
```

So, returning to the `xmonad` configuration we started with, the `keys` field is a function which, given an `XConfig`, produces a keymapping. It uses the monoid of functions to combine the pre-existing function

that generates the keymap to produce as many maps as you have `mappend`ed functions, then combines all the key maps into one.

This part:

```
>> return ()
```

Says that the key assignment is performing some effects and only performing some effects. Functions have to reduce to some result, but sometimes their only purpose is to perform some effects, and you don't want to do anything with the "result" of evaluating the terms.

As we've said and other people have noted as well, monoids are *everywhere*—not just in Haskell but in all of programming.

19.3 Functor

There's a reason we chose that Michael Neale quotation for the `Functor` chapter epigraph: lifting really is the cheat mode. `fmap` is ubiquitous in Haskell, for all sorts of applications, but we've picked a couple that we find especially demonstrative of why it's so handy.

Lifting over IO

This time we're taking a function that doesn't perform I/O, `addUTCTime`, partially applying it to the offset we're going to add to the second argument, then mapping it over the IO action that gets us the current time:

```
import Data.Time.Clock

offsetCurrentTime :: NominalDiffTime
                  -> IO UTCTime
offsetCurrentTime offset =
  fmap (addUTCTime (offset * 24 * 3600)) $
    getCurrentTime
```

Context for the above:

1. `NominalDiffTime` is a newtype of `Pico` and has a `Num` instance—that's why the arithmetic works.

```
addUTCTime :: NominalDiffTime
           -> UTCTime
           -> UTCTime
```

2. `getCurrentTime :: IO UTCTime`

3. `fmap`'s type gets specialized.

```
fmap :: (UTCTime -> UTCTime)
     -> IO UTCTime
     -> IO UTCTime
```

Here, we're lifting some data conversion stuff over the fact that the UUID library has to touch an outside resource (random number generation) to give us a random identifier. The UUID library used is named `uuid` on Hackage. The `Text` package used is named... `text`:

```
import           Data.Text (Text)
import qualified Data.Text as T
import qualified Data.UUID as UUID
import qualified Data.UUID.V4 as UUIDv4

textUuid :: IO Text
textUuid =
  fmap (T.pack . UUID.toString)
       UUIDv4.nextRandom
```

1. `nextRandom :: IO UUID`

2. `toString :: UUID -> String`

3. `pack :: String -> Text`

4. `fmap :: (UUID -> Text)`
   `-> IO UUID`
   `-> IO Text`

Lifting over web app monads

Frequently, when you write web applications, you'll have a custom datatype to describe the entire application, which is also a `Monad`. It's a `Monad`, because your "app context" will have a type parameter to describe what result is produced in the course of a running web application. Often, these types will abstract out the availability of a request or other configuration data with a `Reader` (explained in a later chapter), as well as the performance of effects via `IO`. In the following example, we're lifting over `AppHandler` *and* `Maybe`:

```
userAgent :: AppHandler (Maybe UserAgent)
userAgent =
  (fmap . fmap) userAgent' getRequest

userAgent' :: Request -> Maybe UserAgent
userAgent' req =
  getHeader "User-Agent" req
```

We need the `Functor` here, because, while we can pattern match on the `Maybe` value, an `AppHandler` isn't something we can pattern match on. It's a convention in this web framework library, `snap`, to make a type alias for your web application type. It usually looks like this:

```
type AppHandler = Handler App App
```

The underlying infrastructure for `snap` is more complicated than we can cover to any depth here, but it suffices to say there are a few things floating around:

1. The HTTP request that triggers the current process.
2. The current (possibly empty or default) response that will be returned to the client when the handlers and middleware are done.
3. A function for updating the request timeout.
4. A helper function for logging.
5. And a fair bit more than this.

The issue here is that your `AppHandler` is meant to be slotted into a web application that requires the reading in of a configuration, the initialization of a web server, and the sending of a request to get everything in motion. This is essentially a bunch of functions waiting for arguments—waiting for something to do. It doesn't make sense to do all of that yourself every time you want a value that can only be obtained in the course of the web application doing its thing. Accordingly, our `Functor` is letting us write functions over a structure that handles all this work. It's like we're saying, "Here's a function, apply it to a thing that results from an HTTP request coming down the pipe, if one comes along."

19.4 Applicative

`Applicative` is somewhat new to Haskell, but it's useful enough, particularly with parsers, that it's easy to find examples. There's a whole chapter on parsers coming up later, but we thought these examples were mostly comprehensible even without that context.

hgrev

This is an example from Luke Hoersten's `hgrev` project. The example in the README is a bit dense, but it uses `Monoid` and `Applicative` to combine parsers of command line arguments:

```
jsonSwitch :: Parser (a -> a)
jsonSwitch =

  infoOption $(hgRevStateTH jsonFormat)
  $  long "json"
  <> short 'J'
  <> help
     "Display JSON version information"

parserInfo :: ParserInfo (a -> a)
parserInfo =
  info (helper <*> verSwitch <* jsonSwitch)
       fullDesc
```

You might be wondering what the `<*` operator is. It's another operator from the `Applicative` type class. It allows you to sequence actions, discarding the result of the second argument. Does this look familiar?

```
Prelude> :t (<*)
(<*) :: Applicative f => f a -> f b -> f a

Prelude> :t const
const :: a -> b -> a
```

Basically, the `<*` operator (like its sibling, `*>`, and the monadic operator, `>>`) is useful when you're emitting effects. In this case, you're doing something with effects and want to discard any value that results.

More parsing

Here, we're using `Applicative` to lift the data constructor for the `Payload` type over the `Parser` returned by requesting a value by key out of a JSON object, which is basically an association of text keys to more JSON values, which may be strings, numbers, arrays, or more JSON objects:

```
parseJSON :: Value -> Parser a
(.:) :: FromJSON a
     => Object
     -> Text
     -> Parser a

instance FromJSON Payload where
  parseJSON (Object v) =
    Payload <$> v .: "from"
            <*> v .: "to"
            <*> v .: "subject"
            <*> v .: "body"
            <*> v .: "offset_seconds"
  parseJSON v = typeMismatch "Payload" v
```

This is the same as the JSON example but for CSV[1] data:

```
parseRecord :: Record -> Parser a

instance FromRecord Release where
  parseRecord v
    | V.length v == 5 = Release <$> v .! 0
                                <*> v .! 1
                                <*> v .! 2
                                <*> v .! 3
                                <*> v .! 4
    | otherwise = mzero
```

This one uses `liftA2` to lift the tuple data constructor over `parseKey` and `parseValue` to give key-value pairings. You can see the `<*` operator in there again, as well, along with the infix operator for `fmap` and `=<<`, too:

```
instance Deserializeable ShowInfoResp where
  parser =
    e2err =<< convertPairs
              . HM.fromList <$> parsePairs
    where
      parsePairs :: Parser [(Text, Text)]
      parsePairs =
        parsePair `sepBy` endOfLine

      parsePair =
        liftA2 (,) parseKey parseValue
      parseKey =
        takeTill (==':') <* kvSep

      kvSep = string ": "

      parseValue = takeTill isEndOfLine
```

This one instance is a virtual cornucopia of applications of the previous chapters, and we believe it demonstrates how much cleaner and more readable these structures can make your code.

[1] CSV stands for comma-separated values, a common, though not entirely standardized, file format.

And now for something different

This next example also uses an applicative, but it is a bit different from the examples above. We'll spend more time explaining it, as this pattern for writing utility functions is common:

```
module Web.Shipping.Utils ((<||>)) where

import Control.Applicative (liftA2)

(<||>) :: (a -> Bool)
       -> (a -> Bool)
       -> a
       -> Bool
(<||>) = liftA2 (||)
```

At first glance, this doesn't seem too hard to understand, but some examples will help you develop an understanding of what's going on. Let's start with the operator for Boolean disjunction, `||`, which is an *or* operation:

```
Prelude> True || False
True
Prelude> False || False
False
Prelude> (2 > 3) || (3 == 3)
True
```

And now we want to be able to keep that as an infix operator but lift it over some context, so we use `liftA2`:

```
Prelude> import Control.Applicative
Prelude> (<||>) = liftA2 (||)
```

And we'll make some trivial functions again for the purposes of demonstration:

```
Prelude> f 9001 = True; f _ = False
Prelude> g 42 = True; g _ = False
Prelude> :t f
f :: (Eq a, Num a) => a -> Bool
```

```
Prelude> f 42
False
Prelude> f 9001
True
Prelude> g 42
True
Prelude> g 9001
False
```

We can compose the two functions `f` and `g` to take one input and give one summary result like this:

```
Prelude> (\n -> f n || g n) 0
False
Prelude> (\n -> f n || g n) 9001
True
Prelude> :t (\n -> f n || g n)
(\n -> f n || g n)
  :: (Eq a, Num a) => a -> Bool
```

But we have to pass in that argument `n` in order to do it that way. Our utility function gives us a cleaner way:

```
Prelude> (f <||> g) 0
False
Prelude> (f <||> g) 9001
True
```

It's parallel application of the functions against an argument. That application produces two values, so we monoidally combine the two values so that we have a single value to return. We've set up an environment so that two `a -> Bool` functions that don't have an `a` argument yet can return a result based on those two `Bool` values when the combined function is eventually applied against an `a`.

19.5 Monad

Because effectful programming is constrained in Haskell through the use of `IO`, and `IO` has an instance of `Monad`, examples of `Monad` in practical Haskell code are everywhere. We tried to find some examples that illustrate different interesting use cases.

Opening a network socket

Here, we're using `do` syntax for `IO`'s `Monad` in order to bind a socket handle from the `socket` smart constructor, connect it to an address, then return the handle for reading and writing. This example is from `haproxy-haskell` by Michael Xavier. See the `network` library on Hackage for use and documentation:

```
import Network.Socket

openSocket :: FilePath -> IO Socket
openSocket p = do
  sock <- socket AF_UNIX
                 Stream
                 defaultProtocol
  connect sock sockAddr
  return sock
  where sockAddr =
          SockAddrUnix . encodeString $ p
```

This isn't too unlike anything you saw in previous chapters, at least since we built the hangman game. The next example is a bit richer.

Binding over failure in initialization

Michael Xavier's Seraph is a process monitor and has a `main` entry point that is typical of more developed libraries and applications. The outermost `Monad` is `IO`, but the monad transformer variant of `Either`, called `EitherT`, is used to bind over the possibility of failure in constructing an initialization function. This possibility of failure centers on being able to pull up a correct configuration:

```
main :: IO ()
main = do
  initAndFp <- runEitherT $ do
    fp <- tryHead NoConfig =<< lift getArgs
    initCfg <- load' fp
    return (initCfg, fp)
```

```
  either bail (uncurry boot) initAndFp
  where
    boot initCfg fp =
      void $ runMVC mempty
             oracleModel (core initCfg fp)

    bail NoConfig =
      errorExit "Please pass a config"

    bail (InvalidConfig e) =
      errorExit
        ("Invalid config " ++ show e)

    load' fp =
      hoistEither
      . fmapL InvalidConfig
        =<< lift (load fp)
```

If you found that very dense and difficult to follow at this point, we'd encourage you to have another look at it after we've covered monad transformers.

19.6 An end-to-end example: URL shortener

In this section, we're going to walk through an entire program, beginning to end.[2] There are some pieces we are not going to explain thoroughly—however, this is something you can build and work with, if you're interested in doing so.

First, the `.cabal` file for the project:

```
name:                shawty
version:             0.1.0.0
synopsis:            URI shortener
description:         Please see README.md
homepage:            http://github.com/
license:             BSD3
license-file:        LICENSE
```

[2]The code in this example can be found here: `https://github.com/bitemyapp/shawty-prime/blob/master/app/Main.hs`.

```
author:              Chris Allen
maintainer:          cma@bitemyapp.com
copyright:           2019, Chris Allen
category:            Web
build-type:          Simple
cabal-version:       >=1.10

executable shawty
  hs-source-dirs: app
  main-is:        Main.hs
  ghc-options:    -threaded
  build-depends:  base
                , bytestring
                , hedis
                , mtl
                , network-uri
                , random
                , scotty
                , semigroups
                , text
                , transformers
  default-language:    Haskell2010
```

And the project layout:

```
$ tree
.
├── LICENSE
├── Setup.hs
├── app
│   └── Main.hs
├── shawty.cabal
└── stack.yaml
```

You may choose to use Stack or not. That is how we got the template for the project in place. If you'd like to learn more, check out Stack's GitHub repo[3] and our Stack video mega-tutorial[4] we worked on together. The code following from this point is in `Main.hs`.

[3]You can find it here: `https://github.com/commercialhaskell/stack`.

[4]The whole video is long, but it covers a lot of abnormal use cases. Use the

We need to start our program off with a language extension:

```
{-# LANGUAGE OverloadedStrings #-}
```

`OverloadedStrings` is a way to make `String` literals polymorphic, the way numeric literals are polymorphic over the `Num` type class. `String` literals are not ordinarily polymorphic, as `String` is a concrete type. Using `OverloadedStrings` allows us to use `String` literals as `Text` and `ByteString` values.

Brief aside about polymorphic literals

We mentioned that the integral number literals in Haskell are typed `Num a => a` by default. Now that we have another example to work with, it's worth examining how they work under the hood, so to speak. First, let's look at a type class from a module in `base`:

```
Prelude> import Data.String
Prelude> :info IsString
class IsString a where
  fromString :: String -> a
   -- Defined in Data.String
instance IsString [Char]
-- Defined in Data.String
```

Then, we may notice something in `Num` and `Fractional`:

```
class Num a where
  -- irrelevant bits elided
  fromInteger :: Integer -> a

class Num a => Fractional a where
  -- elision again
  fromRational :: Rational -> a
```

OK, and what about our literals?

---

time stamps to jump to what you need to learn: `https://www.youtube.com/watch?v=sRonIB8ZStw&feature=youtu.be`.

```
Prelude> :set -XOverloadedStrings
Prelude> :t 1
1 :: Num a => a
Prelude> :t 1.0
1.0 :: Fractional a => a
Prelude> :t "blah"
"blah" :: IsString a => a
```

The basic design is that the underlying representation is concrete, but GHC automatically wraps it in `fromString`/`fromInteger`/`fromRational`. So it's as if:

```
{-# LANGUAGE OverloadedStrings #-}

"blah" :: Text
  == fromString ("blah" :: String)

1 :: Int
  == fromInteger (1 :: Integer)

2.5 :: Double
  == fromRational (2.5 :: Rational)
```

Libraries like `text` and `bytestring` provide instances for `IsString` in order to perform the conversion. Assuming you have those libraries installed, you can kick this around a little. Note that, due to the monomorphism restriction, the following will work in the REPL but would not work if we load it from a source file (because it would default to a concrete type; we've seen this a couple of times already, earlier in the book):

```
Prelude> :set -XOverloadedStrings
Prelude> a = "blah"
Prelude> a
"blah"
Prelude> :t a
a :: Data.String.IsString a => a
```

Then, you can make it a `Text` or `ByteString` value:

```
Prelude> import Data.Text (Text)
Prelude> :{
*Main| import Data.ByteString (ByteString)
*Main| :}
Prelude> t = "blah" :: Text
Prelude> bs = "blah" :: ByteString
Prelude> t == bs

Couldn't match expected type ‘Text’ with
  actual type ‘ByteString’
In the second argument of ‘(==)’,
  namely ‘bs’
In the expression: t == bs
```

`OverloadedStrings` is a convenience that originated in the desire of working Haskell programmers to use `String` literals for `Text` and `ByteString` values. It's not too big a deal, but it can be nice and saves you manually wrapping each literal in `fromString`.

Back to the show

Next, the module name must be `Main`, as that is required for anything exporting a `main` executable to be invoked when the program runs. We follow the `OverloadedStrings` extension with our imports:

```
module Main where

import Control.Monad (replicateM)
import Control.Monad.IO.Class (liftIO)
import qualified Data.ByteString.Char8
  as BC
import Data.Text.Encoding
  (decodeUtf8, encodeUtf8)
import qualified Data.Text.Lazy as TL
import qualified Database.Redis as R
import Network.URI (URI, parseURI)
import qualified System.Random as SR
import Web.Scotty
```

Where we import something `qualified`, we are doing two things. Qualifying the import means that we can only refer to values in the module with the full module path, and we use `as` to give the module that we want in scope a name. For example, `Data.ByteString.Char8.pack` is a fully qualified reference to `pack`. We qualify the import so that we don't import declarations that would conflict with bindings that already exist in `Prelude`. By specifying a name using `as`, we can give the value a shorter, more convenient name. Where we import the module name followed by parentheses, such as with `replicateM` or `liftIO`, we are saying we only want to import the functions or values of those names and nothing else. In the case of `import Web.Scotty`, we are importing everything `Web.Scotty` exports. An unqualified and unspecific import should be avoided, except in those cases where the provenance of the imported functions will be obvious, or when the import is a toolkit you must use all together, such as `scotty`.

Next, we need to generate our shortened URLs that will refer to the links people post to the service. We will make a `String` of the characters we want to select from:

```
alphaNum :: String
alphaNum = ['A'..'Z'] ++ ['0'..'9']
```

Now we need to pick random elements from `alphaNum`. The general idea here should be familiar from the hangman game. First, we find the length of the list to determine a range to select from, then get a random number in that range, using `IO` to handle the randomness:

```
randomElement :: String -> IO Char
randomElement xs = do
  let maxIndex :: Int
      maxIndex = length xs - 1
  -- Right of arrow is IO Int,
  -- so randomDigit is Int
  randomDigit <- SR.randomRIO (0, maxIndex)
  return (xs !! randomDigit)
```

Next, we apply `randomElement` to `alphaNum` to get a single random letter or number from our alphabet. Then, we use `replicateM 7` to repeat this action seven times and generate a list of seven random letters or numbers:

```
shortyGen :: IO [Char]
shortyGen =
  replicateM 7 (randomElement alphaNum)
```

For additional fun, see what `replicateM 2 [1, 3]` does and whether you can figure out why. Compare it to the `Prelude` function, `replicate`.

You may have noticed a mention of Redis in our imports and wondered what was up. If you're not already familiar with it, Redis is in-memory, key-value data storage. The details of how Redis works are well beyond the scope of this book, and they're not important here. Redis can be convenient for some common use cases like caching or when you want persistence without a lot of ceremony, as is the case here. You will need to install and have Redis running in order for the project to work; otherwise, the web server will throw an error upon failing to connect to Redis.

This next bit is a function, the arguments of which represent our connection to Redis (`R.Connection`), the key we are setting in Redis, and the value we are setting the key to. We also perform side effects in `IO` to get `Either R.Reply R.Status` as a result. The key in this case is the randomly generated URI we create, and the value is the URL the user wants the shortener to provide at that address:

```
saveURI :: R.Connection
        -> BC.ByteString
        -> BC.ByteString
        -> IO (Either R.Reply R.Status)
saveURI conn shortURI uri =
  R.runRedis conn $ R.set shortURI uri
```

The next function, `getURI`, takes the connection to Redis and the shortened URI key in order to get the URI associated with that short URL and show users where they're headed:

```
getURI :: R.Connection
       -> BC.ByteString
       -> IO (Either R.Reply
              (Maybe BC.ByteString))
getURI conn shortURI =
  R.runRedis conn $ R.get shortURI
```

Next, some basic templating functions for returning output to the web browser:

```
linkShorty :: String -> String
linkShorty shorty =
  concat
  [ "<a href=\""
  , shorty
  , "\">Copy and paste your short URL</a>"
  ]
```

The final output to `scotty` has to be a `Text` value, so we're concatenating lists of `Text` values to produce responses to the browser:

```
-- TL.concat :: [TL.Text] -> TL.Text
shortyCreated :: Show a
              => a
              -> String
              -> TL.Text

shortyCreated resp shawty =
  TL.concat [ TL.pack (show resp)
            , " shorty is: "
            , TL.pack (linkShorty shawty)
            ]

shortyAintUri :: TL.Text -> TL.Text
shortyAintUri uri =
  TL.concat
    [ uri
    , " wasn't a url,"
    , " did you forget http://?"
    ]

shortyFound :: TL.Text -> TL.Text
shortyFound tbs =
  TL.concat
    [ "<a href=\""
    , tbs, "\">"
    , tbs, "</a>" ]
```

Now, we get to the bulk of the web-appy bits in the form of our application. We'll enumerate the application in chunks, but they're all in one `app` function:

```
app :: R.Connection
    -> ScottyM ()

app rConn = do
--  [1]
  get "/" $ do
--  [2]
    uri <- param "uri"
--          [3]
```

1. The Redis connection that should be fired up before the web server starts.

2. `get` is a function that takes a `RoutePattern`, an action that returns an HTTP response, and adds the route to the Scotty server it's embedded in. As you might suspect, `RoutePattern` has an `IsString` instance so that the pattern can be a `String` literal. The top-level route is expressed as `"/"`, i.e., like going to `https://google.com/` or `https://bitemyapp.com/`. That final `/` character is what's being expressed.

3. The `param` function is a means of getting… parameters:

   ```
   param :: Parsable a
         => Data.Text.Internal.Lazy.Text
         -> ActionM a
   ```

   It's sort of like `Read`, but it's parsing a value of the type you ask for. The `param` function can find arguments via URL path captures (see below with `:short`), HTML form inputs, or query parameters. The first argument to `param` is the "name" or key for the input. We cannot explain the entirety of HTTP and HTML here, but the following are means of getting a parameter with the key *name*:

   a) URL path capture

```
      get "/user/:name/view" $ do

    -- requesting the URL /user/Blah/view
    -- would make name = "Blah"
    -- such as:
    -- http://localhost:3000/user/Blah/view
```

b) HTML input form. Here, the name attribute for the input field is `"name"`:

```
<form>
  Name:<br>
  <input type="text" name="name">
</form>
```

c) Query parameters for URIs are pairings of keys and values following a question mark:

```
http://localhost:3000/?name=Blah
```

You can define more than one by using an ampersand to separate the key-value pairs:

```
/?name=Blah&state=Texas
```

Now for the next chunk of the `app` function:

```
  let parsedUri :: Maybe URI
      parsedUri =
        parseURI (TL.unpack uri)

  case parsedUri of
--     [1]
    Just _  -> do
      shawty <- liftIO shortyGen
--              [2]
      let shorty = BC.pack shawty
--                    [3]
```

```
          uri' =
            encodeUtf8 (TL.toStrict uri)
--                  [4]
      resp <-
        liftIO (saveURI rConn shorty uri')
--          [5]
      html (shortyCreated resp shawty)
--      [6]
    Nothing -> text (shortyAintUri uri)
--              [7]
```

1. We test that the user gives us a valid URI by using the `network-uri` library's `parseURI` function. We don't really care about the datatype it gets wrapped in, so when we check if it's `Just` or `Nothing`, we drop it on the floor.

2. The `Monad` here is `ActionM` (an alias of `ActionT`), which is a datatype representing code that handles web requests and returns responses. You can perform `IO` actions in this `Monad`, but you have to lift the `IO` action over the additional structure. Conventionally, one uses `MonadIO` as a sort of auto-lift for `IO` actions, but you could do it manually. We won't demonstrate this here. We will explain monad transformers in a later chapter, so that `ActionT` will eventually be less mysterious.

3. Converting the short code for the URI into a `Char8 ByteString` for storage in Redis.

4. Converting the URI the user provides from a lazy `Text` value into a strict `Text` value, then encoding it as a UTF-8 (a common Unicode format) `ByteString` for storage in Redis.

5. Again, using `liftIO` so that we can perform an `IO` action inside a `scotty ActionM`. In this case, we're saving the short code and the URI in Redis so that we can look things up with the short code as a key, then get the URI back as a value if it has been stored in the past.

6. The templated response we return when we successfully save the short code for the URI. This gives the user a shortened URI to share.

7. Error response in case the user gives us a URI that isn't valid.

The second handler handles requests to a shortened URI and returns the unshortened URL to follow:

```
  get "/:short" $ do
--    [1]
    short <- param "short"
--           [2]
    uri <- liftIO (getURI rConn short)
--              [3]

    case uri of
      Left reply ->
        text (TL.pack (show reply))
--    [4]         [5]
      Right mbBS -> case mbBS of
--    [6]
        Nothing -> text "uri not found"
--      [7]
        Just bs -> html (shortyFound tbs)

--         [8]
          where tbs :: TL.Text
                tbs =
                TL.fromStrict
                  (decodeUtf8 bs)
--                [9]
```

1. This is the URL path capture we mentioned earlier, such that requesting `/blah` from the server will cause it to get the key `"blah"` from Redis and, if there's a value stored in that key, return that URI in the response. To do that in a web browser or on the command line (with, for example, `curl` or `wget`), you'd point your client at `http://localhost:3000/blah` to test it.

2. Same parameter fetching as before. This time we expect it to be part of the path capture rather than a query argument.

3. Lifting an `IO` action inside `ActionM` again, this time to get the short code as the lookup key from Redis.
4. `Left` here (in the `Either` we get back from Redis) signifies some kind of failure, usually an error.
5. `Text` response returning an error in case we get `Left`, so that the user knows what the error is, taking advantage of Redis having `Show`able errors to render it in the response.
6. Happy path.
7. Just because an error doesn't happen doesn't mean the key is in the database.
8. We fetch a key that exists in the database, get the `ByteString` out of the `Just` data constructor, and render the URI in the success template to show the user the URI that we're storing.
9. Going in the opposite direction we went in before—decoding the `ByteString` on the assumption that it's encoded as UTF-8, then converting from a strict `Text` value to a lazy `Text` value.

Now, we come to the `main` event. `main` returns `IO ()` and acts as the entry point for our web server when we start the executable. We begin by invoking `scotty 3000`, a helper function from the `scotty` framework which, given a port to run on and a Scotty application, will listen for requests and respond to them:

```
main :: IO ()
main = do
  rConn <- R.connect R.defaultConnectInfo
  scotty 3000 (app rConn)
```

And that is the entirety of this URL shortener. We have a couple of exercises based on this code, and we encourage you to come back to them after we've covered monad transformers, as well, to see how your comprehension is growing.

Exercise

In the URL shortener, an important step was omitted. We're not checking if we're overwriting an existing short code, which is entirely possible despite them being randomly generated. We can calculate the odds of this by examining the cardinality of the values:

```
-- alphaNum = ['A'..'Z'] ++ ['0'..'9']
-- shortyGen =
--   replicateM 7 (randomElement alphaNum)

length alphaNum ^ 7 == 78364164096
```

So, the problem is, what if we accidentally clobber a previously generated short URI? There are a few ways of solving this. One is to check to see if the short URI already exists in the database before saving it and throwing an error if it does. This is going to be vanishingly unlikely to happen unless you've suddenly become a very popular URI shortening service, but it'd prevent the loss of any data. Your exercise is to devise some means of making this less likely. The easiest way would be to simply make the short codes long enough that you'd need to run a computer until the heat death of the universe to get a collision, but you should try throwing an error in the first handler we showed you.

19.7 That's a wrap!

We hope this chapter gave you some idea of how Haskellers use the type classes we've been talking about in real code, to handle various types of problems. In the next two chapters, we'll be looking at `Foldable` and `Traversable`, two type classes with some interesting properties that rely on these algebraic structures, so we encourage you to take some time to explore the use cases we demonstrated here. Consider going back to anything you didn't understand well the first time you went through the chapters on `Monoid`, `Functor`, `Applicative`, and `Monad`.

19.8 Follow-up resources

1. Bryan O'Sullivan. *The case of the mysterious explosion in space*. Explains how GHC handles string literals.
   `http://www.serpentine.com/blog/2012/09/12/the-case-of-the-mysterious-explosion-in-space/`

Chapter 20

Foldable

You gotta know when to hold
'em, know when to fold 'em,
know when to walk away,
know when to run.

Kenny Rogers

20.1 Foldable

This type class has been appearing in type signatures at least since Chapter 3, but for your purposes in those early chapters, we said you could think of a `Foldable` thing as a list. As you saw in Chapter 10, on folding lists, lists are foldable data structures. But it is also true that lists are not the only foldable data structures, so this chapter will expand on the idea of catamorphisms and generalize it to many datatypes.

A list fold is a way to reduce the values inside a list to one summary value by recursively applying some function. It is sometimes difficult to appreciate that, as filtering and mapping functions may be implemented in terms of a fold and yet return an entirely new list! The new list is the summary value of the old list after being reduced, or transformed, by function application.

The folding function is always dependent on some `Monoid` instance. The folds we wrote previously mostly relied on implicit monoidal operations. As we'll see in this chapter, generalizing catamorphisms to other datatypes depends on understanding the monoids for those structures and, in some cases, making them explicit.

This chapter will cover:

- The `Foldable` class and its core operations.
- The monoidal nature of folding.
- Standard operations derived from folding.

20.2 The Foldable class

The Hackage documentation for the `Foldable` type class describes it as being a "class of data structures that can be folded to a summary value." The folding operations that we've seen previously fit neatly into that definition, but this type class includes many operations. We're going to go through the full definition a little at a time. The definition in the library begins like this:

```
class Foldable t where
    {-# MINIMAL foldMap | foldr #-}
```

The MINIMAL annotation on the type class tells you that a minimally complete definition of it must define either foldMap *or* foldr for a given datatype. As it happens, foldMap and foldr can each be implemented in terms of the other, and the other operations included in the type class can be implemented in terms of either of them. As long as at least one is defined, you have a working instance of Foldable. Some methods in the type class have default implementations that can be overridden when needed. This is in case there's a more efficient way to do something that's specific to your datatype.

If you query the info about the type class in GHCi, the first line of the definition includes the kind signature for t:

```
class Foldable (t :: * -> *) where
```

That t should be a higher-kinded type is not surprising: lists are higher-kinded types. We need t to be a type constructor for the same reasons we did with Functor, and we will see that the effects are similar. Types that take more than one type argument, such as tuples and Either, will necessarily have their first type argument included as part of their structure.

Please note that you will need to use GHC 7.10 or later versions for all the examples in this chapter to work. Also, while the Prelude as of GHCi 7.10 includes many changes related to the Foldable type class, not all of Foldable is in the Prelude. To follow along with the examples in the chapter, you may need to import Data.Foldable and Data.Monoid (for some of the Monoid newtypes).

20.3 Revenge of the monoids

One thing we did not talk about when we covered folds previously is the importance of monoids. Folding necessarily implies a binary associative operation that has an identity value. The first two operations defined in Foldable make this explicit:

```
class Foldable (t :: * -> *) where
  fold :: Monoid m => t m -> m
  foldMap :: Monoid m
          => (a -> m) -> t a -> m
```

While `fold` allows you to combine elements inside a `Foldable` structure using the `Monoid` defined for those elements, `foldMap` first maps each element of the structure to a `Monoid` and then combines the results using that instance of `Monoid`.

These might seem a little weird until you realize that `Foldable` requires that you make the implicit `Monoid` visible in folding operations. Let's take a look at a basic `foldr` operation and see how it compares to `fold` and `foldMap`:

```
Prelude> foldr (+) 0 [1..5]
15
```

The binary associative operation for this fold is `+`, so we've specified it without thinking of it as a monoid. The fact that the numbers in our list have other possible monoids is not relevant once we've specified which operation to use.

We can already see from the type of `fold` that it's not going to work in the same way as `foldr`, because it doesn't take a function for its first argument. But we also can't just fold up a list of numbers, because the `fold` function doesn't specify a `Monoid`:

```
Prelude> fold (+) [1, 2, 3, 4, 5]
-- error message resulting from incorrect
-- number of arguments

Prelude> fold [1, 2, 3, 4, 5]
-- error message resulting from not having
-- an instance of Monoid
```

So, to make `fold` work, we need to specify a `Monoid` instance:

```
Prelude> xs = map Sum [1..5]
Prelude> fold xs
Sum {getSum = 15}
```

Or, less tediously:

```
Prelude> :{
*Main| let xs :: [Sum Integer]
*Main|     xs = [1, 2, 3, 4, 5]
```

```
*Main| :}
Prelude> fold xs
Sum {getSum = 15}
Prelude> :{
*Main| let xs :: [Product Integer]
*Main|     xs = [1, 2, 3, 4, 5]
*Main| :}
Prelude> fold xs
Product {getProduct = 120}
```

In some cases, the compiler can identify and use the standard `Monoid` for a type, without requiring us to be explicit:

```
Prelude> foldr (++) "" ["hello", " julie"]
"hello julie"
Prelude> fold ["hello", " julie"]
"hello julie"
```

The default `Monoid` instance for lists gives us what we need, so we don't have to specify it ourselves.

And now for something different

Let's turn our attention now to `foldMap`. Unlike `fold`, `foldMap` has a function as its first argument. Unlike `foldr`, the first (function) argument of `foldMap` must explicitly map each element of a foldable structure to a `Monoid`:

```
Prelude> foldMap Sum [1, 2, 3, 4]
Sum {getSum = 10}
Prelude> foldMap Product [1, 2, 3, 4]
Product {getProduct = 24}

Prelude> foldMap All [True, False, True]
All {getAll = False}
Prelude> foldMap Any [(3 == 4), (9 > 5)]
Any {getAny = True}

Prelude> xs = [Just 1, Nothing, Just 5]
Prelude> foldMap First xs
```

```
First {getFirst = Just 1}
Prelude> foldMap Last xs
Last {getLast = Just 5}
```

In the above examples, the function being applied is a data constructor. The data constructor identifies the `Monoid` instance—the `mappend`—for those types. It already contains enough information to allow `foldMap` to reduce the collection of values to one summary value.

However, `foldMap` can also have a function to map that is different from the `Monoid` it's using:

```
Prelude> xs = map Product [1..3]
Prelude> foldMap (*5) xs
Product {getProduct = 750}
-- 5 * 10 * 15
750

Prelude> xs = map Sum [1..3]
Prelude> foldMap (*5) xs
Sum {getSum = 30}
-- 5 + 10 + 15
30
```

It can map the function to each value first and then use the `Monoid` instance to reduce them to one value. Compare this to `foldr`, where the function has the `Monoid` instance baked in:

```
Prelude> foldr (*) 5 [1, 2, 3]
-- (1 * (2 * (3 * 5)))
30
```

In fact, due to the way `foldr` works, declaring a `Monoid` instance that is different from what is implied in the folding function doesn't change the final result:

```
Prelude> sumXs = map Sum [2..4]
Prelude> foldr (*) 3 sumXs
Sum {getSum = 72}
Prelude> productXs = map Product [2..4]
```

```
Prelude> foldr (*) 3 productXs
Product {getProduct = 72}
```

However, it is worth pointing out that if what you're trying to fold only contains one value, declaring a `Monoid` instance won't change the behavior of `foldMap`, either:

```
Prelude> fm = foldMap (*5)
Prelude> fm (Just 100) :: Product Integer
Product {getProduct = 500}
Prelude> fm (Just 5) :: Sum Integer
Sum {getSum = 25}
```

With only one value, it doesn't need the `Monoid` instance. Specifying the `Monoid` instance is necessary to satisfy the type checker, but with only one value, there is nothing to `mappend`. It just applies the function. It will use the `mempty` value from the declared `Monoid` instance, though, in cases where what you are trying to fold is empty:

```
Prelude> fm Nothing :: Sum Integer
Sum {getSum = 0}
Prelude> fm Nothing :: Product Integer
Product {getProduct = 1}
```

So, what we've seen so far is that `Foldable` is a way of generalizing a catamorphism—a folding—to different datatypes, and at least in some cases, it forces you to think about the monoid you're using to combine values.

20.4 Demonstrating Foldable instances

As we said above, a minimal `Foldable` instance must have either `foldr` or `foldMap`. Any of the other functions in this type class can be derived from one or the other of those. With that said, let's turn our attention to implementing `Foldable` instances for different types.

Identity

We'll kick things off by writing a `Foldable` instance for `Identity`:

```
data Identity a =
  Identity a
```

We're only obligated to write `foldr` *or* `foldMap`, but we'll write both, plus `foldl` so you get the gist of it:

```
instance Foldable Identity where
  foldr f z (Identity x) = f x z

  foldl f z (Identity x) = f z x

  foldMap f (Identity x) = f x
```

With `foldr` and `foldl`, we're doing basically the same thing, but with the arguments swapped. We didn't need to do anything special for `foldMap`.

It may seem strange to think of folding one value. When we discussed catamorphisms previously, we focused on how they can reduce a bunch of values down to one summary value. In the case of this `Identity` catamorphism, though, the point is less to reduce the values inside the structure to one value and more to consume, or use, the value:

```
Prelude> foldr (*) 1 (Identity 5)
5
Prelude> foldl (*) 5 (Identity 5)
25

Prelude> fm = foldMap (*5)
Prelude> type PI = Product Integer
Prelude> fm (Identity 100) :: PI
Product {getProduct = 500}
```

Maybe

This one is a little more interesting because, unlike with `Identity`, we have to account for the `Nothing` cases. When the `Maybe` value that we're folding is `Nothing`, we need to be able to return some "zero" value, while doing nothing with the folding function but also disposing of

the `Maybe` structure. For `foldr` and `foldl`, that zero value is the start value provided:

```
Prelude> foldr (+) 1 Nothing
1
```

On the other hand, for `foldMap`, we use the `Monoid`'s identity value as our zero:

```
Prelude> fm = foldMap (+1)
Prelude> fm Nothing :: Sum Integer
Sum {getSum = 0}
```

When the value is a `Just` value, though, we need to apply the folding function to it and, again, dispose of the structure:

```
Prelude> foldr (+) 1 (Just 3)
4
Prelude> fm $ Just 3 :: Sum Integer
Sum {getSum = 4}
```

So, let's look at the instance. We'll use a fake `Maybe` type again, to avoid conflict with the `Maybe` instance that already exists:

```
instance Foldable Optional where
  foldr _ z Nada = z
  foldr f z (Yep x) = f x z

  foldl _ z Nada = z
  foldl f z (Yep x) = f z x

  foldMap _ Nada = mempty
  foldMap f (Yep a) = f a
```

Note that if you don't tell the compiler which `Monoid` you mean, it will complain about the type being ambiguous:

```
Prelude> foldMap (+1) Nada

No instance for (Num a0) arising
  from a use of ‘it’
```

```
The type variable ‘a0’ is ambiguous
...blah blah who cares...
```

So, we need to assert a type that has a `Monoid` for this to work:

```
Prelude> import Data.Monoid
Prelude> foldMap (+1) Nada :: Sum Int
Sum {getSum = 0}
Prelude> foldMap (+1) Nada :: Product Int
Product {getProduct = 1}

Prelude> foldMap (+1) (Just 1) :: Sum Int
Sum {getSum = 2}
```

With a `Nada` value and a declared type of `Sum Int` (giving us our `Monoid`), `foldMap` gives us `Sum 0`, because that is the `mempty`, or identity, for `Sum`. Similarly with `Nada` and `Product`, we get `Product 1`, because that is the identity for `Product`.

20.5 Some basic derived operations

The `Foldable` type class includes some other operations that we haven't covered in this context yet. Some of these, such as `length`, were previously defined for use with lists, but their types have been generalized now to make them useful with other types of data structures. Below are descriptions, type signatures, and examples for several of these:

```
-- | List of elements of a structure,
-- from left to right.
toList :: t a -> [a]


Prelude> toList (Just 1)
[1]
Prelude> xs = [Just 1, Just 2, Just 3]
Prelude> map toList xs
[[1],[2],[3]]
Prelude> concatMap toList xs
[1,2,3]
```

```
Prelude> xs = [Just 1, Just 2, Nothing]
Prelude> concatMap toList xs
[1,2]
Prelude> toList (1, 2)
[2]
```

Why doesn't it put the 1 in the list? For the same reason that `fmap` doesn't apply a function to the 1.

```
-- | Test whether the structure is empty.
null :: t a -> Bool
```

Notice that `null` returns `True` on `Left` and `Nothing` values, just as it does on empty lists and so forth:

```
Prelude> null (Left 3)
True
Prelude> null []
True
Prelude> null Nothing
True
Prelude> null (1, 2)
False
Prelude> xs = [Just 1, Just 2, Nothing]
Prelude> fmap null xs
[False,False,True]
```

The next one, `length`, returns a count of how many `a` values inhabit the `t a`. In a list, that could be multiple `a` values due to the definition of that datatype. It's important to note, though, that for tuples, the first argument (as well as the leftmost, or outermost, type argument of datatypes such as `Maybe` and `Either`) is part of the `t` here, not part of the `a`.

```
-- | Returns the size/length of a finite
-- structure as an 'Int'.
length :: t a -> Int
```

```
Prelude> length (1, 2)
1
Prelude> xs = [(1, 2), (3, 4), (5, 6)]
Prelude> length xs
3
Prelude> fmap length xs
[1,1,1]

Prelude> fmap length Just [1, 2, 3]
1
```

That last example looks strange—we know. But if you run it in your REPL, you'll see that it returns the result we promise. Why? And why does this return the same result?

```
Prelude> length $ Just [1, 2, 3]
1
```

The `a` of `Just a` in the last case above is a list.

```
Prelude> xs = [Just 1, Just 2, Just 3]
Prelude> fmap length xs
[1,1,1]
Prelude> xs = [Just 1, Just 2, Nothing]
Prelude> fmap length xs
[1,1,0]
```

```
-- | Does the element occur
--   in the structure?
elem :: Eq a => a -> t a -> Bool
```

We've used `Either` in the following example set to demonstrate the behavior of `Foldable` functions with `Left` values. As we saw with `Functor`, you can't map over the left data constructor, because the left type argument is part of the structure. In the following example set, that means that `elem` can't see inside the `Left` constructor to whatever the value is, so the result will be `False`, even if the value matches:

```
Prelude> elem 2 (Just 3)
False
```

```
Prelude> elem True (Left False)
False
Prelude> elem True (Left True)
False
Prelude> elem True (Right False)
False
Prelude> elem True (Right True)
True
Prelude> xs = [Right 1,Right 2,Right 3]
Prelude> fmap (elem 3) xs
[False,False,True]
```

```
-- | The largest element
--   of a non-empty structure.
maximum :: Ord a => t a -> a

-- | The least element
--   of a non-empty structure.
minimum :: Ord a => t a -> a
```

Here, notice that `Left` and `Nothing` (and similar) values are *empty* for the purposes of these functions:

```
Prelude> maximum [10, 12, 33, 5]
33
Prelude> xs = [Just 2, Just 10, Just 4]
Prelude> fmap maximum xs
[2,10,4]
Prelude> fmap maximum (Just [3, 7, 10, 2])
Just 10

Prelude> minimum "julie"
'e'
Prelude> fmap minimum (Just "julie")
Just 'e'
Prelude> xs = map Just "jul"
Prelude> xs
[Just 'j',Just 'u',Just 'l']
Prelude> fmap minimum xs
```

```
"jul"

Prelude> xs = [Just 4, Just 3, Nothing]
Prelude> fmap minimum xs
[4,3,*** Exception:
          minimum: empty structure
Prelude> minimum (Left 3)
*** Exception: minimum: empty structure
```

We've seen `sum` and `product` before, and they do what their names suggest: return the sum and product of the members of a structure:

```
sum :: (Foldable t, Num a) => t a -> a

product :: (Foldable t, Num a) => t a -> a
```

And now for some examples:

```
Prelude> sum (7, 5)
5
Prelude> fmap sum [(7, 5), (3, 4)]
[5,4]
Prelude> fmap sum (Just [1, 2, 3, 4, 5])
Just 15
Prelude> product Nothing
1
Prelude> fmap product (Just [])
Just 1
Prelude> fmap product (Right [1, 2, 3])
Right 6
```

Exercises: Library functions

Implement the functions in terms of `foldMap` or `foldr` from `Foldable`, then try them out with multiple types that have `Foldable` instances.

1. This and the next one are nicer with `foldMap`, but the `foldr` function is fine, too:

   ```
   sum :: (Foldable t, Num a) => t a -> a
   ```

2. ```
   product :: (Foldable t, Num a) => t a -> a
   ```

3. ```
   elem :: (Foldable t, Eq a)
        => a -> t a -> Bool
   ```

4. ```
   minimum :: (Foldable t, Ord a)
           => t a -> Maybe a
   ```

5. ```
   maximum :: (Foldable t, Ord a)
           => t a -> Maybe a
   ```

6. ```
   null :: (Foldable t) => t a -> Bool
   ```

7. ```
   length :: (Foldable t) => t a -> Int
   ```

8. Some say this is all `Foldable` amounts to:

   ```
   toList :: (Foldable t) => t a -> [a]
   ```

9. Hint: use `foldMap`:

   ```
   -- | Combine the elements
   --   of a structure using a monoid.
   fold :: (Foldable t, Monoid m) => t m -> m
   ```

10. Define `foldMap` in terms of `foldr`:

    ```
    foldMap :: (Foldable t, Monoid m)
            => (a -> m) -> t a -> m
    ```

20.6 Chapter exercises

Write `Foldable` instances for the following datatypes:

1. ```
   data Constant a b =
     Constant b
   ```

2. ```
   data Two a b =
     Two a b
   ```

3. ```
   data Three a b c =
     Three a b c
   ```
   ```

4. ```
   data Three' a b =
     Three' a b b
   ```

5. ```
   data Four' a b =
     Four' a b b b
   ```

Thinking cap time. Write a filter function for `Foldable` types using the `foldMap` function:

```
filterF :: ( Applicative f
           , Foldable t
           , Monoid (f a))
        => (a -> Bool) -> t a -> f a
filterF = undefined
```

20.7 Follow-up resources

1. Jakub Arnold. *Foldable and Traversable*.

Chapter 21

Traversable

> O, thou hast damnable iteration and art indeed able to corrupt a saint.
>
> William Shakespeare

21.1 Traversable

`Functor` gives us a way to transform values embedded in a structure. `Applicative` gives us a way to transform values contained within a structure using a function that is also embedded in a structure. This means that each application produces the effect of "adding structure," which is then applicatively combined with other such effects. `Foldable` gives us a way to process values embedded in a structure as if they existed in a sequential order, as we've seen ever since we learned about list folding.

`Traversable` was introduced in the same paper as `Applicative`, and its introduction to `Prelude` likewise didn't come until the release of GHC 7.10. However, it was available as part of the `base` library before that. `Traversable` depends on `Applicative`, and thus `Functor`, and is also superclassed by `Foldable`.

`Traversable` allows you to transform elements inside a structure like a functor, producing applicative effects along the way, and lift those potentially multiple instances of applicative structure outside of the traversable structure. It is commonly described as a way to traverse a data structure, mapping a function inside a structure while accumulating applicative contexts in the process. This is easiest to see, perhaps, through the liberal demonstration of examples, so let's get to it.

In this chapter, we will:

- Explain the `Traversable` type class and its canonical functions.
- Explore examples of `Traversable` in practical use.
- Tidy up some code using this type class.
- And, of course, write some `Traversable` instances.

21.2 The Traversable type class definition

This is the type class definition as it appears in the library `Data.Traversable`:

```
class (Functor t, Foldable t)
   => Traversable t where
    traverse :: Applicative f
             => (a -> f b)
             -> t a
             -> f (t b)
    traverse f = sequenceA . fmap f
```

The `traverse` function maps each element of a structure to an action, evaluates the actions from left to right, and collects the results. If you find yourself with a value that has a type like `[IO a]`, it's possible you made a mistake and used `fmap`, where you actually need `traverse`. Unless a list of `IO` actions that can be independently dispatched is really what you want. You'd do something like this with `traverse`:

```
fmap :: Functor f
     => (a -> b)
     -> f a
     -> f b

traverse :: Applicative f
         => (a -> f b)
         -> t a
         -> f (t b)

myData :: [String]
myFunc :: String -> IO Record

wrong :: [IO Record]
wrong = fmap myFunc myData

right :: IO [Record]
right = traverse myFunc myData
```

When you get comfortable with Haskell, you'll learn to recognize that if you use `fmap` and get a type `t (f b)`, when what you want is `f (t b)`, you instead need to use `Traversable`.

The counterpart to `traverse` is `sequenceA`:

```
  -- | Evaluate each action in the
  -- structure from left to right,
  -- and collect the results.
  sequenceA :: Applicative f
            => t (f a) -> f (t a)
  sequenceA = traverse id
  {-# MINIMAL traverse | sequenceA #-}
```

A minimal instance for this type class provides an implementation of either `traverse` or `sequenceA`, because as you can see, they can be defined in terms of each other. As with `Foldable`, we can define more than the bare minimum, if we can leverage information specific to our datatype to make the behavior more efficient. We're going to focus on these two functions in this chapter, though, as those are the most typically useful.

21.3 sequenceA

We will start with some examples of `sequenceA` in action, as it is the simpler of the two. You can see from the type signature above that the effect of `sequenceA` is to flip two contexts or structures. It doesn't by itself allow you to apply any function to the `a` value inside the structure. It only flips the layers of structure around. Compare these:

```
Prelude> sum [1, 2, 3]
6
Prelude> fmap sum [Just 1, Just 2, Just 3]
[1,2,3]
Prelude> (fmap . fmap) sum Just [1, 2, 3]
Just 6
Prelude> xs = [Just 1, Just 2, Nothing]
Prelude> fmap product xs
[1,2,1]
```

To these:

```
Prelude> fmap Just [1, 2, 3]
[Just 1,Just 2,Just 3]
Prelude> sequenceA $ fmap Just [1, 2, 3]
```

```
Just [1,2,3]
Prelude> xs = [Just 1, Just 2, Just 3]
Prelude> sequenceA xs
Just [1,2,3]
Prelude> xsn = [Just 1, Just 2, Nothing]
Prelude> sequenceA xsn
Nothing
Prelude> fmap sum $ sequenceA xs
Just 6
Prelude> fmap product (sequenceA xsn)
Nothing
```

In the first example, using `sequenceA` flips the structures around—instead of a list of `Maybe` values, you get a `Maybe` of a list value. In the next two examples, we can lift a function over a `Maybe` structure and apply it to the list that is inside, if we have a `Just a` value after applying the `sequenceA`.

It's worth mentioning here that the `Data.Maybe` module has a function called `catMaybes` that offers a different way of handling a list of `Maybe` values:

```
Prelude> import Data.Maybe
Prelude> xs = [Just 1, Just 2, Just 3]
Prelude> catMaybes xs
[1,2,3]
Prelude> xsn = [Just 1, Just 2, Nothing]
Prelude> catMaybes xsn
[1,2]
Prelude> xsn' = xs ++ [Nothing]
Prelude> sum $ catMaybes xsn'
6
Prelude> fmap sum $ sequenceA xsn'
Nothing
```

Using `catMaybes` allows you to sum (or otherwise process) a list of `Maybe` values, even if there's potentially a `Nothing` value lurking within.

21.4 traverse

Let's look next at the type of `traverse`:

```
traverse
  :: (Applicative f, Traversable t)
  => (a -> f b) -> t a -> f (t b)
```

You might notice a similarity between this definition and the types of `fmap` and `=<<` (aka *flip bind*):

```
fmap     :: (a -> b)   -> f a -> f b
(=<<)    :: (a -> m b) -> m a -> m b
traverse :: (a -> f b) -> t a -> f (t b)
```

We're still mapping a function over some embedded value(s), like `fmap`, but as with *flip bind*, that function is itself generating more structure. However, unlike with *flip bind*, that structure can be of a different type than the structure we lift over to apply the function. And at the end, it will flip the two structures around, as `sequenceA` does.

In fact, as we saw in the type class definition, `traverse` is `fmap` composed with `sequenceA`:

```
traverse f = sequenceA . fmap f
```

Let's look at how that works in practice:

```
Prelude> fmap Just [1, 2, 3]
[Just 1,Just 2,Just 3]
Prelude> sequenceA $ fmap Just [1, 2, 3]
Just [1,2,3]
Prelude> sequenceA . fmap Just $ [1, 2, 3]
Just [1,2,3]
Prelude> traverse Just [1, 2, 3]
Just [1,2,3]
```

We'll run through some longer examples in a moment, but the general idea is that anytime you're using `sequenceA . fmap f`, you can use `traverse` to achieve the same result in one step.

mapM is traverse

You may have seen a slightly different way of expressing `traverse` before, in the form of `mapM`.

In versions of GHC prior to 7.10, the type of `mapM` was the following:

```
mapM :: Monad m
     => (a -> m b) -> [a] -> m [b]

-- contrast with

traverse :: Applicative f
         => (a -> f b) -> t a -> f (t b)
```

We can think of `traverse` in `Traversable` as abstracting the `[]` in `mapM` to any traversable data structure and generalizing the `Monad` requirement to only need an `Applicative`. This is valuable, as it means we can use this pattern more widely and with more code. For example, the list datatype is fine for small pluralities of values, but in more performance-sensitive code, you may want to use a `Vector` from the `vector`[1] library. With `traverse`, you won't have to change your code, because the primary `Vector` datatype has a `Traversable` instance and so should work.

Similarly, the type for `sequence` in GHC versions prior to 7.10 is a less useful `sequenceA`:

```
sequence :: Monad m
         => [m a]
         -> m [a]

-- contrast with

sequenceA :: (Applicative f, Traversable t)
          => t (f a)
          -> f (t a)
```

Again, we're generalizing the list to any `Traversable` and weakening the `Monad` requirement to `Applicative`.

[1] `http://hackage.haskell.org/package/vector`

21.5 So, what's Traversable for?

In a literal sense, anytime you need to flip two type constructors around, or map something and then flip them around, that's probably a use case for `Traversable`:

```
sequenceA :: Applicative f
          => t (f a) -> f (t a)

traverse :: Applicative f
         => (a -> f b) -> t a -> f (t b)
```

We'll kick around some hypothetical functions and values without bothering to implement them in the REPL to see when we may want `traverse` or `sequenceA`:

```
Prelude> f = undefined :: a -> Maybe b
Prelude> xs = undefined :: [a]
Prelude> :t map f xs
map f xs :: [Maybe b]
```

But what if we want a value of type `Maybe [b]`? The following will work, but we can do better:

```
Prelude> :t sequenceA $ map f xs
sequenceA $ map f xs :: Maybe [a]
```

It's usually better to use `traverse` whenever we see a `sequence` or `sequenceA` combined with a `map` or `fmap`:

```
Prelude> :t traverse f xs
traverse f xs :: Maybe [b]
```

Next, we'll look at real examples of when you'd want to do this.

21.6 Morse code revisited

We're going to call back to what we did in Chapter 14, on testing, with the Morse code program, in order to look at a nice example of how to use `traverse`. Let's recall what we did there:

```
stringToMorse :: String -> Maybe [Morse]
stringToMorse s =
  sequence $ fmap charToMorse s

-- what we want to do
stringToMorse :: String -> Maybe [Morse]
stringToMorse = traverse charToMorse
```

Normally, you might expect that if you mapped an `(a -> f b)` over a `t a`, you'd end up with `t (f b)`, but `traverse` flips that around. Remember, we had each character conversion wrapped in a `Maybe` due to the possibility of getting characters in a string that aren't translatable into Morse code (or, in the opposite direction, aren't Morse characters):

```
Prelude> morseToChar "gobbledegook"
Nothing
Prelude> morseToChar "-.-."
Just 'c'
```

We can use `fromMaybe` to remove the `Maybe` layer:

```
Prelude> import Data.Maybe
Prelude> fromMaybe ' ' (morseToChar "-.-.")
'c'

Prelude> stringToMorse "chris"
Just ["-.-.","....",".-.","..","..."]

Prelude> import Data.Maybe
Prelude> fromMaybe [] (stringToMorse "chris")
["-.-.","....",".-.","..","..."]
```

We'll define a little helper for use in the following examples:

```
Prelude> morse s = fromMaybe [] (stringToMorse s)
Prelude> :t morse
morse :: String -> [Morse]
```

Now, if we `fmap morseToChar`, we get a list of `Maybe` values:

```
Prelude> fmap morseToChar (morse "chris")
[Just 'c',Just 'h',Just 'r',Just 'i',Just 's']
```

We don't want `catMaybes` here, because it drops the `Nothing` values. What we want is for the presence of *any* `Nothing` value whatsoever to make the final result `Nothing`. The function that gives us what we want for this is `sequence`. We did use `sequence` in the original version of the `stringToMorse` function. `sequence` is useful for flipping your types around, as well (note the positions of the `t` and `m`):

```
sequence :: (Monad m, Traversable t)
         => t (m a) -> m (t a)
```

To use `sequence` over a list of `Maybe` (or other monadic) values, we need to compose it with `fmap`:

```
Prelude> :t (sequence .) . fmap
(sequence .) . fmap
  :: (Monad m, Traversable t) =>
     (a1 -> m a) -> t a1 -> m (t a)

Prelude> sequence $ fmap morseToChar (morse "chris")
Just "chris"

Prelude> sequence $ fmap morseToChar (morse "julie")
Just "julie"
```

The weird looking composition, which you've possibly also seen in the form of `(join .) . fmap` is because `fmap` takes *two* (not *one*) arguments, so the expressions aren't proper unless we compose twice to await a second argument to which to apply `fmap`:

```
-- we want this
(sequence .) . fmap =
  \f xs -> sequence (fmap f xs)

-- not this
sequence . fmap =
  \f -> sequence (fmap f)
```

This composition of `sequence` and `fmap` is so common that `traverse` is now a standard `Prelude` function. Compare the above to this:

```
*Morse T> traverse morseToChar (morse "chris")
Just "chris"

*Morse T> traverse morseToChar (morse "julie")
Just "julie"
```

So, `traverse` is just `fmap` and the `Traversable` version of `sequence` bolted together into one convenient function. `sequence` is the unique bit, but you need to do the `fmap` first, most of the time, so you end up using `traverse`. This is similar to the way `>>=` is just `join` composed with `fmap`, where `join` is the bit that is unique to `Monad`.

21.7 Axing tedious code

Try to bear with us for a moment, and realize that the following is real but also intentionally fake code. That is, one of the authors helped somebody with refactoring their code, and this simplified version is what your author was given. One of the strengths of Haskell is that we can sometimes work in terms of types without worrying about code that actually runs. This code is from Alex Petrov—thanks for the great example, Alex!

```
data Query     = Query
data SomeObj   = SomeObj
data IoOnlyObj = IoOnlyObj
data Err       = Err
```

There's a decoder function that makes some object from a string:

```
decodeFn :: String -> Either Err SomeObj
decodeFn = undefined
```

And there's a query that runs against a database and returns an array of strings:

```
fetchFn :: Query -> IO [String]
fetchFn = undefined
```

We also have an additional "context initializer" that performs `IO`:

```
makeIoOnlyObj :: [SomeObj]
              -> IO [(SomeObj, IoOnlyObj)]
makeIoOnlyObj = undefined
```

Putting them all together:

```
pipelineFn
  :: Query
  -> IO (Either Err [(SomeObj, IoOnlyObj)])
pipelineFn query = do
  a <- fetchFn query
  case sequence (map decodeFn a) of
    (Left  err)  -> return $ Left err
    (Right res) -> do
      a <- makeIoOnlyObj res
      return $ Right a
```

The objective was to clean this up. A few things aroused suspicion:

1. The use of `sequence` and `map`.
2. Manually casing on the result of the `sequence` and the `map`.
3. Binding monadically over the `Either` only to perform another monadic IO action inside of that.

We pared the pipeline function down to this:

```
pipelineFn
  :: Query
  -> IO (Either Err [(SomeObj, IoOnlyObj)])
pipelineFn query = do
  a <- fetchFn query
  traverse makeIoOnlyObj (mapM decodeFn a)
```

Thanks to IRC user merijn for helping with this. We can even make it point-free, if we want to:

```
pipelineFn
  :: Query
  -> IO (Either Err [(SomeObj, IoOnlyObj)])
pipelineFn =
  ((traverse makeIoOnlyObj
  . mapM decodeFn) =<<) . fetchFn
```

And since `mapM` is just `traverse` with a slightly different type:

```
pipelineFn
  :: Query
  -> IO (Either Err [(SomeObj, IoOnlyObj)])
pipelineFn =
  ((traverse makeIoOnlyObj
  . traverse decodeFn) =<<) . fetchFn
```

This is the terse, clean style many Haskellers prefer. As we said back when we first introduced it, point-free style can help focus attention on the functions, rather than on the specifics of the data that are being passed around as arguments. Using functions like `traverse` cleans up code by drawing attention to the ways the types change and signaling the programmer's intent.

21.8 Do all the things

We're going to use an HTTP client library named `wreq`[2] for this demonstration, so we can make calls to a handy-dandy website for testing HTTP clients at `http://httpbin.org/`. Feel free to experiment and substitute your own ideas for HTTP services or websites you can poke and prod:

```
module HttpStuff where

import Data.ByteString.Lazy hiding (map)
import Network.Wreq
```

[2] `http://hackage.haskell.org/package/wreq`

```
-- replace with other websites
-- if desired or needed
urls :: [String]
urls = [ "http://httpbin.org/ip"
       , "http://httpbin.org/bytes/5"
       ]

mappingGet :: [IO (Response ByteString)]
mappingGet = map get urls
```

But what if we don't want a list of IO actions we can perform to get a response, but rather one big IO action that produces a list of responses? This is where `Traversable` can be helpful:

```
traversedUrls :: IO [Response ByteString]
traversedUrls = traverse get urls
```

We hope that these examples have helped demonstrate that `Traversable` is a useful type class. While `Foldable` seems trivial, it is a necessary superclass of `Traversable`, and `Traversable`, like `Functor` and `Monad`, is now widely used in everyday Haskell code, due to its practicality.

Strength for understanding

`Traversable` is stronger than `Functor` and `Foldable`. Because of this, we can recover the `Functor` and `Foldable` instance for a type from the `Traversable`, just as we can recover the `Functor` and `Applicative` from the `Monad`. Here, we can use the `Identity` type to get something that is essentially just `fmap` all over again:

```
Prelude> import Data.Functor.Identity
Prelude> traverse (Identity . (+1)) [1, 2]
Identity [2,3]
Prelude> runIdentity $ traverse (Identity . (+1)) [1, 2]
[2,3]
Prelude> :{
Prelude| let edgeMap f t =
Prelude|       runIdentity $ traverse (Identity . f) t
Prelude| :}
```

```
Prelude> :t edgeMap
edgeMap :: Traversable t => (a -> b) -> t a -> t b
Prelude> edgeMap (+1) [1..5]
[2,3,4,5,6]
```

Using `Const` or `Constant`, we can also recover a `foldMap`-like `Foldable`:

```
Prelude> import Data.Monoid
-- from `transformers`
Prelude> import Data.Functor.Constant

Prelude> xs' = [1, 2, 3, 4, 5]
Prelude> xs = xs' :: [Sum Integer]
Prelude> traverse (Constant . (+1)) xs
Constant (Sum {getSum = 20})

Prelude> :{
Prelude| let foldMap' f t =
Prelude|       getConstant
Prelude|       $ traverse (Constant . f) t
Prelude| :}
Prelude> :t foldMap'
foldMap' :: (Traversable t, Monoid a)
         => (a1 -> a) -> t a1 -> a
Prelude> :t foldMap
foldMap :: (Foldable t, Monoid m)
        => (a -> m) -> t a -> m
```

Doing exercises like this can help strengthen your intuition for the relationships of these type classes and their canonical functions. We know it sometimes feels like these things are pure intellectual exercise, but getting comfortable with manipulating functions like these is ultimately the key to getting comfortable with Haskell. This is how you learn to play type Tetris with the pros.

21.9 Traversable instances

You knew this was coming.

Either

The `Traversable` instance that follows here is identical to the one in the `Data.Traversable` module in `base`, but we've added a `Functor`, `Foldable`, and `Applicative`, so that you can see the progression:

```
data Either a b =
    Left a
  | Right b
  deriving (Eq, Ord, Show)

instance Functor (Either a) where
    fmap _ (Left x) = Left x
    fmap f (Right y) = Right (f y)

instance Applicative (Either e) where
    pure          = Right
    Left  e <*> _ = Left e
    Right f <*> r = fmap f r

instance Foldable (Either a) where
    foldMap _ (Left _) = mempty
    foldMap f (Right y) = f y

    foldr _ z (Left _) = z
    foldr f z (Right y) = f y z

instance Traversable (Either a) where
    traverse _ (Left x) = pure (Left x)
    traverse f (Right y) = Right <$> f y
```

Given what you've seen above, this hopefully isn't too surprising. We have function application and type flipping, in an `Either` context.

Tuple

As above, we've provided a progression of instances, but for the 2-tuple or anonymous product:

```
instance Functor ((,) a) where
  fmap f (x,y) = (x, f y)
```

```
instance Monoid a
      => Applicative ((,) a) where
    pure x = (mempty, x)
    (u, f) <*> (v, x) =
      (u `mappend` v, f x)

instance Foldable ((,) a) where
    foldMap f (_, y) = f y
    foldr f z (_, y) = f y z

instance Traversable ((,) a) where
    traverse f (x, y) = (,) x <$> f y
```

Here, we have much the same, but for a tuple context.

21.10 Traversable laws

The `traverse` function must satisfy the following laws:

1. Naturality

   ```
   t . traverse f = traverse (t . f)
   ```

   This law tells us that function composition behaves in unsurprising ways with respect to a traversed function. Since a traversed function `f` generates the structure that appears on the "outside" of the `traverse` operation, there's no reason we can't float a function over the structure into the traversal itself.

2. Identity

   ```
   traverse Identity = Identity
   ```

   This law states that traversing the data constructor of the `Identity` type over a value will produce the same result as just putting the value in `Identity`. This tells us that `Identity` represents a structural identity for traversing data. This is another way of saying that a `Traversable` instance cannot add or inject any structure or effects.

3. Composition

```
traverse (Compose . fmap g . f) =
  Compose . fmap (traverse g) . traverse f
```

This law demonstrates how we can collapse sequential traversals into a single traversal, by taking advantage of the `Compose` datatype, which combines structure.

The `sequenceA` function must satisfy the following laws:

1. Naturality

   ```
   t . sequenceA = sequenceA . fmap t
   ```

2. Identity

   ```
   sequenceA . fmap Identity = Identity
   ```

3. Composition

   ```
   sequenceA . fmap Compose =
     Compose . fmap sequenceA . sequenceA
   ```

None of this should be too surprising, given what you've seen with `traverse`.

21.11 Quality control

Great news! You can `QuickCheck` your `Traversable` instances, since they have laws! Conveniently, the `checkers` library we've already been using has recorded the laws for us. You can add the following code to a module and change the type alias to determine which instances to test:

```
type TI = []

main = do
  let trigger :: TI (Int, Int, [Int])
      trigger = undefined
  quickBatch (traversable trigger)
```

21.12 Chapter exercises

Traversable instances

Write a `Traversable` instance for the datatype provided, filling in any required superclasses. Use `QuickCheck` to validate your instances.

Identity

Write a `Traversable` instance for `Identity`:

```
newtype Identity a = Identity a
    deriving (Eq, Ord, Show)

instance Traversable Identity where
  traverse = undefined
```

Constant

```
newtype Constant a b =
  Constant { getConstant :: a }
```

Maybe

```
data Optional a =
    Nada
  | Yep a
```

List

```
data List a =
    Nil
  | Cons a (List a)
```

Three

```
data Three a b c =
  Three a b c
```

Pair

```
data Pair a b =
  Pair a b
```

Big

When you have more than one value of type `b`, use `Monoid` and `Applicative` for the `Foldable` and `Traversable` instances, respectively:

```
data Big a b =
  Big a b b
```

Bigger

Same as for `Big`:

```
data Bigger a b =
  Bigger a b b b
```

S

This may be difficult. To make it easier, we'll give you the constraints and `QuickCheck` instances:

```
{-# LANGUAGE FlexibleContexts #-}

module SkiFree where

import Test.QuickCheck
import Test.QuickCheck.Checkers

data S n a = S (n a) a deriving (Eq, Show)

instance ( Functor n
         , Arbitrary (n a)
         , Arbitrary a )
        => Arbitrary (S n a) where
  arbitrary =
    S <$> arbitrary <*> arbitrary
```

```
instance ( Applicative n
         , Testable (n Property)
         , Eq a
         , Eq (n a)
         , EqProp a)
        => EqProp (S n a) where
  (=-=) = eq

instance Traversable n
      => Traversable (S n) where
  traverse = undefined

main =
  sample' (arbitrary :: Gen (S [] Int))
```

Instances for Tree

This might be hard. Write the following instances for `Tree`:

```
data Tree a =
    Empty
  | Leaf a
  | Node (Tree a) a (Tree a)
  deriving (Eq, Show)

instance Functor Tree where
  fmap = undefined

-- foldMap is a bit easier
-- and looks more natural,
-- but you can do foldr, too,
-- for extra credit.
instance Foldable Tree where
  foldMap = undefined

instance Traversable Tree where
  traverse = undefined
```

Hints:

1. For `foldMap`, think `Functor` but with some `Monoid` thrown in.
2. For `traverse`, think `Functor` but with some `Functor`[3] thrown in.

21.13 Follow-up resources

1. Jakub Arnold. *Foldable and Traversable*.
2. Jeremy Gibbons and Bruno C. d. S. Oliveira. *The Essence of the Iterator Pattern*.
3. Conor McBride and Ross Paterson. *Applicative programming with effects*.

[3] Not a typo.

Chapter 22

Reader

> The tears of the world are a constant quantity. For each one who begins to weep somewhere else another stops. The same is true of the laugh.
>
> Samuel Beckett

22.1 Reader

The last two chapters were focused on some type classes that might still seem strange and difficult to you. The next three chapters are going to focus on some patterns that might still seem strange and difficult. `Foldable`, `Traversable`, `Reader`, `State`, and parser combinators are not strictly necessary to understanding and using Haskell. We do have reasons for introducing them now, but those reasons might not seem clear to you for a while. If you don't quite grasp all of it on the first pass, that's completely fine. Read it through, do your best with the exercises, and then come back when you feel like you're ready.

When writing applications, programmers often need to pass around some information that may be needed intermittently or universally throughout an entire application. We don't want to simply pass this information as arguments, because it would be present in the type of almost every function. This can make the code harder to read and harder to maintain. To address this, we use `Reader`.

In this chapter, we will:

- Examine the `Functor`, `Applicative`, and `Monad` type class instances for *functions*.
- Learn about the `Reader` newtype.
- See some examples of using `Reader`.

22.2 A new beginning

We're going to set this chapter up a bit differently from previous chapters, because we're hoping to demonstrate that what we're doing here is not *that* different from things you've done before. So, we're going to start with some examples. Start a file like this:

```
import Control.Applicative

boop = (*2)
doop = (+10)

bip :: Integer -> Integer
bip = boop . doop
```

We know that the `bip` function will take one argument because of the types of `boop`, `doop`, and the `(.)` operator. Note that if you do not specify the types and load `bip` from a file, it will be monomorphic by default. If you wish to make `bip` polymorphic, you may change its signature, but you also need to specify a polymorphic type for the two functions from which it's built. The rest of the chapter will wait while you verify these things.

When we apply `bip` to an argument, `doop` will be applied to that argument first, and then the result of that will be passed as input to `boop`. So far, nothing new.

We can also write the composition of these functions like this:

```
bloop :: Integer -> Integer
bloop = fmap boop doop
```

We aren't accustomed to mapping a function over another function, and you may be wondering what the functorial context here is. By "functorial context," we mean the structure (datatype) that we must lift the function *over* in order to apply it to the value inside. For example, a list is a functorial context we can lift functions over. We say that the function gets lifted over the structure of the list and applied to or mapped over the values that are inside the list.

In `bloop`, the context is a partially applied function. As in function composition, `fmap` composes the two functions before applying them to the argument. The result of the one can then get passed to the next as input. Using `fmap` here lifts the one partially applied function over the next, in a sense setting up something like this:

```
fmap boop doop x == (*2) ((+10) x)
```

When this `x` comes along, it's the first required argument to `(+10)`, and then the result for that is the first required argument to `(*2)`.

This is the `Functor` of functions. We're going to go into more detail about it soon.

For now, let's turn to another set of examples. Put these in the same file, so `boop` and `doop` are still in scope:

```
bbop :: Integer -> Integer
bbop = (+) <$> boop <*> doop

duwop :: Integer -> Integer
duwop = liftA2 (+) boop doop
```

Now we're in an `Applicative` context. We're adding another function to lift over the contexts of our partially applied functions. This time, we still have partially applied functions that are awaiting application to an argument, but this works differently than using `fmap` does. This time, the argument gets passed to both `boop` and `doop` in parallel, and the results are added together.

`boop` and `doop` are each waiting for an input. We can apply them both at once like this:

```
Prelude> bbop 3
19
```

That does something like this:

```
((+) <$> (*2) <*> (+10)) 3

-- First the fmap
(*2) :: Num a => a -> a
(+)  :: Num a => a -> a -> a
(+) <$> (*2) :: Num a => a -> a -> a
```

Mapping a function awaiting two arguments over a function awaiting one produces a two argument function.

Remember, this is identical to function composition:

```
(+) . (*2) :: Num a => a -> a -> a
```

With the same result:

```
Prelude> ((+) . (*2)) 5 3
13
Prelude> ((+) <$> (*2)) 5 3
13
```

So what's happening?

```
((+) <$> (*2)) 5 3

-- Keeping in mind that this is
-- the (.) operator under the hood
((+) . (*2)) 5 3

-- f . g = \x -> f (g x)

((+) . (*2)) == \x -> (+) (2 * x)
```

The tricky part here is that even after we apply `x`, we've got the `+` operator partially applied to the first argument, which is doubled by `(*2)`. There's a second argument, and that's what will get added to the first argument that gets doubled.

The first function to get applied is `(*2)`, and the first argument is *5*. `(*2)` takes one argument, so we get:

```
((+) . (*2)) 5 3
(\x -> (+) (2 * x)) 5 3
(\5 -> (+) (2 * 5)) 3
((+) 10) 3

-- Then it adds 10 and 3
13
```

OK, but what about the second bit?

```
((+) <$> (*2) <*> (+10)) 3

-- Wait, what? What happened to the
-- first argument?
((+) <$> (*2) <*> (+10)) :: Num b => b -> b
```

One of the nice things about Haskell is that we can assert a more concrete type for functions like `<*>` and see if the compiler agrees that we're putting forth something hypothetically possible. Let's remind ourselves of the type of `<*>`:

```
Prelude> :t (<*>)
(<*>) :: Applicative f =>
  f (a -> b) -> f a -> f b
```

In this case, we know `f` is `((->) a)`, so we concretize it thusly:

```
appReader :: (a -> a -> b)
          -> (a -> a)
          -> (a -> b)
appReader = (<*>)
```

The compiler agrees that this is a possible type for `<*>`.

So how does that work? What's happening is we're feeding a single argument to `(*2)` and `(+10)`, and the two results form the two arguments to `+`:

```
((+) <$> (*2) <*> (+10)) 3
```

```
(3*2) + (3+10)
```

```
6 + 13
```

```
19
```

We'd use this when two functions share the same input, and we want to apply some other function to their result in order to reach a final result. This happens more than you might think, and we saw an example of it back in Chapter 19, on applying structure:

```
module Web.Shipping.Utils ((<||>)) where

import Control.Applicative (liftA2)

(<||>) :: (a -> Bool)
       -> (a -> Bool)
       -> a
       -> Bool
(<||>) = liftA2 (||)
```

This is the same idea as `duwop`, above.

Finally, another example:

```
boopDoop :: Integer -> Integer
boopDoop = do
    a <- boop
    b <- doop
    return (a + b)
```

This will do precisely the same thing as the `Applicative` example, but this time the context is monadic. This distinction doesn't much matter with this particular function. We assign the variable `a` to the partially applied function `boop`, and `b` to `doop`. As soon as we receive an input, it will fill the empty slots in `boop` and `doop`. The results will be bound to the variables `a` and `b` and passed into `return`.

So, we've seen here that we can have a `Functor`, `Applicative`, and `Monad` for partially applied functions. In all cases, these are awaiting application to one argument that will allow both functions to be evaluated. The `Functor` of functions is function composition. The `Applicative` and `Monad` chain the argument forward in addition to the composition (applicatives and monads are both varieties of functors, so they retain that core functorial behavior).

This is the idea of `Reader`. It is a way of stringing functions together when all those functions are awaiting one input from a shared environment. We're going to get into the details of how it works, but the important intuition here is that it's another way of abstracting out function application, and it gives us a way to do computation in terms of an argument that hasn't been supplied yet. We use this most often when we have a constant value that we will obtain from somewhere outside our program that will be an argument to a whole bunch of functions. Using `Reader` allows us to avoid passing that argument around explicitly.

Short Exercise: Warming up

We'll be doing something here similar to what you saw above, to give you practice and help you try to develop a feel or intuition for what is to come. These are similar enough to what you just saw that you can almost copy and paste, so try not to overthink them too much.

First, start a file off like this:

```
import Data.Char

cap :: [Char] -> [Char]
cap xs = map toUpper xs

rev :: [Char] -> [Char]
rev xs = reverse xs
```

Two simple functions with the same type, taking the same type of input. We could compose them, using the `(.)` operator or `fmap`:

```
composed :: [Char] -> [Char]
composed = undefined

fmapped :: [Char] -> [Char]
fmapped = undefined
```

The output of these two functions should be identical, one string that is made all uppercase and reversed, like this:

```
Prelude> composed "Julie"
"EILUJ"
Prelude> fmapped "Chris"
"SIRHC"
```

We want to return the results of `cap` and `rev` as a tuple, like this:

```
Prelude> tupled "Julie"
("JULIE","eiluJ")
-- or
Prelude> tupled' "Julie"
("eiluJ","JULIE")
```

We want to use an `Applicative` here. The type looks like this:

```
tupled :: [Char] -> ([Char], [Char])
```

There is no special reason such a function needs to be monadic, but let's do that, too, to get some practice. Do it one time using `do` syntax. Then try writing a new version using `>>=`. The types will be the same as the type for `tupled`.

22.3 This is Reader

As we saw above, functions have `Functor`, `Applicative`, and `Monad` instances. Usually when you see or hear the term `Reader`, it'll be referring to the `Monad` instance.

We use function composition, because it lets us compose two functions without explicitly having to recognize the argument that will eventually arrive—the `Functor` of functions is function composition. With the `Functor` of functions, we are able to map an ordinary function over another to create a new function awaiting a final argument. The `Applicative` and `Monad` instances for the function type give us a way to map a function that is awaiting an `a` over another function that is also awaiting an `a`.

Giving it a name helps us know the what and why of what we're doing: reading an argument from the environment into functions. It'll be especially nice for clarity's sake later when we make the `ReaderT` *monad transformer*.

Exciting, right? Let's back up here and go into more detail about how `Reader` works.

22.4 Breaking down the Functor of functions

If you type `:info Functor` in your REPL, one of the instances you might notice is the one for the partially applied type constructor of functions `((->) r)`:

```
instance Functor ((->) r)
```

This instance can be a little confusing, so we're going to unwind it until it's a bit more comfortable. First, let's see what we can accomplish with it:

```
Prelude> fmap (+1) (*2) 3
7
```

Rearranging a little bit:

```
Prelude> fmap (+1) (*2) $ 3
7
```

```
Prelude> (fmap (+1) (*2)) 3
7
```

This should look familiar:

```
Prelude> (+1) . (*2) $ 3
7

Prelude> (+2) . (*1) $ 2
4

Prelude> fmap (+2) (*1) $ 2
4

Prelude> (+2) `fmap` (*1) $ 2
4
```

Fortunately, there's nothing weird going on here. If you check the implementation of the instance in `base`, you'll find the following:

```
instance Functor ((->) r) where
    fmap = (.)
```

Let's unravel the types. Remember that `->` takes two arguments and therefore has the kind `* -> * -> *`. So, we know upfront that we have to apply one of the type arguments before we can have a `Functor`. With the `Either Functor`, we know that we will lift over the `Either a`, and if our function is applied, it will be applied to the `b` value. With the function type:

```
data (->) a b
```

The same rule applies: you have to lift over the `(->) a` and only transform the `b` value. The `a` is conventionally called `r` for `Reader` in these instances, but a type variable of any other name smells as sweet. Here, `r` is the first argument of `(a -> b)`:

```
-- Type constructor of functions
(->)
-- Fully applied
a -> b

((->) r)
-- is
r ->
```

So `r`, above, is the type of the argument to the function.

From this, we can determine that `r`, the argument type for functions, is part of the structure being *lifted over* when we lift over a function, not the value being transformed or mapped over.

This leaves the result of the function as the value being transformed. This happens to line up neatly with what function composition is about:

```
(.) :: (b -> c) -> (a -> b) -> a -> c
```

Or perhaps:

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
```

Now, how does this line up with `Functor`?

```
(.)  :: (b -> c) -> (a -> b) -> (a -> c)

fmap :: Functor f => (a -> b) -> f a -> f b
```

We'll remove the names of the functions and the type class constraint, as we can take them for granted from here on out:

```
:: (b -> c) -> (a -> b) -> (a -> c)
:: (a -> b) ->     f a  ->     f b
```

Changing up the letters without changing the meaning:

```
:: (b -> c) -> (a -> b) -> (a -> c)
:: (b -> c) ->     f b  ->     f c

-- f is ((->) a)
```

```
:: (b -> c)
->      (a -> b)
->      (a -> c)
:: (b -> c)
-> ((->) a)   b
-> ((->) a)   c
```

Unroll the prefix notation into infix:

```
:: (b -> c) -> (a -> b) -> (a -> c)
:: (b -> c) -> (a -> b) -> (a -> c)
```

Bada bing. Functorial lifting for functions.

22.5 But uh, Reader?

Ah yes, right. `Reader` is a newtype wrapper for the function type:

```
newtype Reader r a =
  Reader { runReader :: r -> a }
```

The `r` is the type we're reading in, and `a` is the result type of our function.

The `Reader` newtype has a handy `runReader` accessor to get the function out of `Reader`. Let us prove for ourselves that this is the same thing but with a touch of data constructor jiggery-pokery mixed in. What does the `Functor` for this look like, compared to function composition?

```
instance Functor (Reader r) where
  fmap :: (a -> b)
       -> Reader r a
       -> Reader r b

  fmap f (Reader ra) =
    Reader $ \r -> f (ra r)

-- same as (.)
compose :: (b -> c) -> (a -> b) -> (a -> c)
compose f g = \x -> f (g x)
```

```
-- see it?
  \r -> f (ra r)
  \x -> f (g  x)
```

Basically, the same thing right? In the `Reader` functor, `ra` has the type `r -> a`, and `f` has the type `a -> b`. Applying `ra` to the value `r` yields a value of type `a`, to which `f` is then applied, yielding a value of type `b`. Function composition!

We can use the fact that we recognize this as function composition to make a slightly different instance for `Reader`:

```
instance Functor (Reader r) where
   fmap :: (a -> b)
        -> Reader r a
        -> Reader r b
  fmap f (Reader ra) =
    Reader $ (f . ra)
```

So what we're doing here is basically:

1. Unpack `r -> a` out of `Reader`.
2. Compose `f` with the function we unpack out of `Reader`.
3. Put the new, composed function back into `Reader`.

Without the `Reader` newtype, we drop steps 1 and 3 and have function composition.

Exercise: Ask

Implement the following function. If you get stuck, remember it's less complicated than it looks. Write down what you *know*. What do you know about the type `a`? What does the type simplify to? How many inhabitants does that type have? You've seen this type before:

```
ask :: Reader a a
ask = Reader ???
```

22.6 Functions have an Applicative, too

We've seen a couple of examples already of the `Applicative` of functions and how it works. Now, we'll get into the details.

The first thing we want to do is notice how the types specialize:

```
-- Applicative f =>

-- f ~ (->) r

pure :: a ->     f a
pure :: a -> (r -> a)

(<*>) ::    f (a -> b)
         ->     f a
         ->     f b

(<*>) :: (r -> a -> b)
         -> (r -> a)
         -> (r -> b)
```

As we saw in the `Functor` instance, the `r` of `Reader` is part of the `f` structure. We have two arguments in this function, and both of them are functions waiting for the `r` input. When that comes, both functions will be applied to return a final result of `b`.

Demonstrating the function Applicative

This example is similar to other demonstrations we've done previously in the book, but this time we'll be aiming to show you what specific use the `Applicative` of functions typically has. We start with some newtypes for tracking our different `String` values:

```
newtype HumanName =
  HumanName String
  deriving (Eq, Show)

newtype DogName =
  DogName String
  deriving (Eq, Show)
```

```
newtype Address =
  Address String
  deriving (Eq, Show)
```

We do this so that our types are more self-explanatory, to express intent, and so we don't accidentally mix up our inputs. A type like:

```
String -> String -> String
```

Is difficult when:

1. They aren't strictly *any* string values.
2. They aren't processed in an identical fashion. You don't handle addresses in the same way as names.

So make the difference explicit.
We'll make two record types:

```
data Person =
  Person {
    humanName :: HumanName
  , dogName :: DogName
  , address :: Address
  } deriving (Eq, Show)

data Dog =
  Dog {
    dogsName :: DogName
  , dogsAddress :: Address
  } deriving (Eq, Show)
```

The following are sample data for us to use. You can modify them however you want:

```
pers :: Person
pers =
  Person (HumanName "Big Bird")
         (DogName "Barkley")
         (Address "Sesame Street")
```

```
chris :: Person
chris =
  Person (HumanName "Chris Allen")
         (DogName "Papu")
         (Address "Austin")
```

And here is how we'd write it both with and without `Reader`:

```
-- without Reader
getDog :: Person -> Dog
getDog p =
  Dog (dogName p) (address p)

-- with Reader
getDogR :: Person -> Dog
getDogR =
  Dog <$> dogName <*> address
```

Can't see the `Reader`? What if we concrete up the types a bit?

```
(<$->>) :: (a -> b)
        -> (r -> a)
        -> (r -> b)
(<$->>) = (<$>)

(<*->>) :: (r -> a -> b)
        -> (r -> a)
        -> (r -> b)
(<*->>) = (<*>)

-- with Reader
getDogR' :: Person -> Dog
getDogR' =
  Dog <$->> dogName <*->> address
```

What we're trying to highlight here is that *Reader* is not always `Reader`—sometimes it's the ambient `Applicative` or `Monad` associated with the partially applied function type, here that is `r ->`.

The pattern of using `Applicative` in this manner is common, so there's an alternative way to do this, using `liftA2`:

```
import Control.Applicative (liftA2)

-- with Reader, alternative
getDogR' :: Person -> Dog
getDogR' =
  liftA2 Dog dogName address
```

Here's the type of `liftA2`:

```
liftA2 :: Applicative f =>
          (a -> b -> c)
       -> f a -> f b -> f c
```

Again, we're waiting for an input from elsewhere. Rather than having to thread the argument through our functions, we elide it and let the types manage it for us.

Exercise: Reading comprehension

1. Write `liftA2` yourself. Think about it in terms of abstracting out the difference between `getDogR` and `getDogR'`, if that helps.

   ```
   myLiftA2 :: Applicative f =>
               (a -> b -> c)
            -> f a -> f b -> f c
   myLiftA2 = undefined
   ```

2. Write the following function. Again, it is simpler than it looks.

   ```
   asks :: (r -> a) -> Reader r a
   asks f = Reader ???
   ```

3. Implement the `Applicative` for `Reader`.

   To write the `Applicative` instance for `Reader`, we'll use an extension called `InstanceSigs`. It's an extension we need in order to assert types for type class methods. You ordinarily cannot assert type signatures in instances. The compiler already knows the types of the functions, so it's not usually necessary to assert the types in instances anyway. We're doing this for the sake of clarity, to make the `Reader` type explicit in our signatures:

```
-- you'll need this pragma
{-# LANGUAGE InstanceSigs #-}

instance Applicative (Reader r) where
  pure :: a -> Reader r a
  pure a = Reader $ ???

  (<*>) :: Reader r (a -> b)
        -> Reader r a
        -> Reader r b
  (Reader rab) <*> (Reader ra) =
    Reader $ \r -> ???
```

Some instructions and hints.

a) When writing the `pure` function for `Reader`, remember that what you're trying to construct is a function that takes a value of type `r`, which you know nothing about, and returns a value of type `a`. Given that you're not really doing anything with `r`, there's *really only one thing you can do*.

b) We got the definition of the apply function started for you, so we'll describe what you need to do, and you write the code. If you unpack the type of `Reader`'s apply operation above, you get the following:

```
<*> :: (r -> a -> b)
    -> (r -> a)
    -> (r -> b)
```

Contrast this with the type of `fmap`:

```
fmap :: (a -> b)
     -> (r -> a)
     -> (r -> b)
```

So what's the difference? The difference is that apply, unlike `fmap`, also takes an argument of type `r`.

Make it so.

22.7 The Monad of functions

Functions also have a `Monad` instance. You saw this in the beginning of the chapter, and you perhaps have some intuition now for how it must work. We're going to walk through a simplified demonstration of how it works before we get to the types and instance definition. Feel free to work through this section as quickly or as slowly as you think appropriate for your own grasp of what we've presented so far.

Let's start by supposing that we could write a couple of functions, like so:

```
foo :: (Functor f, Num a) => f a -> f a
foo r = fmap (+1) r

bar :: Foldable f => t -> f a -> (t, Int)
bar r t = (r, length t)
```

Now, as it happens in our program, we want to make one function that will do both things—increment the values inside our structure and also tell us the length of the value. We could write that like this:

```
froot :: Num a => [a] -> ([a], Int)
froot r = (map (+1) r, length r)
```

Or we could write the same function by combining the two functions we already have. As it is written above, `bar` takes two arguments. We could write a version that takes only one argument, so that both parts of the tuple apply to the same argument. That is easy enough to do (notice the change in the type signature, as well):

```
barOne :: Foldable t => t a -> (t a, Int)
barOne r = (r, length r)
```

That gives us the reduction to one argument that we want but doesn't increment the values in the list as our `foo` function does. We can add that, this way:

```
barPlus r = (foo r, length r)
```

But we can also do it more compactly by making `(foo r)` the first argument to `bar`:

```
frooty :: Num a => [a] -> ([a], Int)
frooty r = bar (foo r) r
```

Now, we have an environment in which two functions are waiting for the same argument to come in. They'll both apply to that argument in order to produce a final result.

Let's make a small change to make it look a little more `Reader`-y:

```
frooty' :: Num a => [a] -> ([a], Int)
frooty' = \r -> bar (foo r) r
```

Then we abstract it out, so that it's not specific to these functions:

```
fooBind m k = \r -> k (m r) r
```

In this polymorphic version, the type signature will look like this:

```
fooBind :: (t2 -> t1)
        -> (t1 -> t2 -> t)
        -> t2
        -> t
```

So many `t` types! That's because we can't know very much about those types once our function is that abstract. We can make it a little clearer by making some substitutions. We'll use the `r` to represent the argument that both functions are waiting on—the `Reader`-y part:

```
fooBind :: (r -> a)
        -> (a -> r -> b)
        -> (r -> b)
```

If we could take the `r` parts out, we might notice that `fooBind` itself looks like a very abstract and simplified version of something we've seen before (overparenthesizing a bit, for clarity):

```
(>>=) :: Monad m =>
  m    a   -> (a -> (m    b))  ->  m    b
 (r -> a)  -> (a -> (r -> b))  -> (r -> b)
```

This is how we get to the `Monad` of functions. Just as with the `Functor` and `Applicative` instances, the `((->) r)` is our structure—the `m` in the type of `>>=`. In the next section, we'll work forward from the types.

The Monad instance

As we noted, the `r` argument remains part of our (monadic) structure:

```
(>>=) :: Monad m
      => m a -> (a -> m b)      ->      m b
(>>=) ::
    (->) r a -> (a -> (->) r b) -> (->) r b

(>>=) ::
    (r -> a) -> (a ->   r -> b) ->   r -> b

return :: Monad m => a ->      m a
return ::            a -> (->) r a
return ::            a ->   r -> a
```

You may notice that `return` looks like a function we've seen *a lot* of in this book.

Let's look at it side by side with the `Applicative`:

```
(<*>) :: (r -> a -> b)
      -> (r -> a)
      -> (r -> b)

(>>=) :: (r -> a)
      -> (a -> r -> b)
      -> (r -> b)
```

Or with the flipped bind:

```
(<*>) :: (r -> a -> b)
      -> (r -> a)
      -> (r -> b)

(=<<) :: (a -> r -> b)
      -> (r -> a)
      -> (r -> b)
```

So you've got this ever-present type `r` following your functions around like a lonely puppy.

Example uses of the Reader type

Remember the earlier example with `Person` and `Dog`? Here's the same but with the `Reader Monad` and `do` syntax:

```
getDogRM :: Person -> Dog
getDogRM = do
  name <- dogName
  addy <- address
  return $ Dog name addy
```

Exercise: Reader Monad

1. Implement the `Reader Monad`.

   ```
   -- Don't forget InstanceSigs.
   instance Monad (Reader r) where
     return = pure

     (>>=) :: Reader r a
           -> (a -> Reader r b)
           -> Reader r b
     (Reader ra) >>= aRb =
       Reader $ \r -> ???
   ```

   Hint: constrast the type with the `Applicative` instance, and perform the most obvious change you can imagine in order to make it work.

2. Rewrite the monadic `getDogRM` to use your `Reader` datatype.

22.8 Reader Monad by itself is boring

It can't do anything the `Applicative` cannot:

```
{-# LANGUAGE NoImplicitPrelude #-}

module PrettyReader where

flip :: (a -> b -> c) -> (b -> a -> c)
flip f a b = f b a
```

```
const :: a -> b -> a
const a b = a

(.) :: (b -> c) -> (a -> b) -> (a -> c)
f . g = \a -> f (g a)

class Functor f where
  fmap :: (a -> b) -> f a -> f b

class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b

class Applicative f => Monad f where
  return :: a -> f a

  (>>=) :: f a -> (a -> f b) -> f b

instance Functor ((->) r) where
  fmap = (.)

instance Applicative ((->) r) where
  pure = const

  f <*> a = \r -> f r (a r)

instance Monad ((->) r) where
  return = pure
  m >>= k = flip k <*> m
```

Speaking generally, in terms of the algebras alone, you *cannot* get a `Monad` instance from the `Applicative`. You can get an `Applicative` from the `Monad`. However, our instances above aren't written in terms of an abstract datatype—we *know* it's the type of functions. Because it's not hiding behind a `Reader` newtype, we can use `flip` and `apply` to make the `Monad` instance. We need specific type information to augment what the `Applicative` is capable of, before we can get our `Monad` instance.

22.9 You can only change what comes below

The "read only" nature of the type argument `r` means that you can swap in a different type or value of `r` for functions that you call, but not for functions that call you. The best way to demonstrate this is by using the `withReaderT` function, which lets us start a new `Reader` context, and providing a different argument:

```
withReaderT
    :: (r' -> r)
    -- ^ The function to modify
    --   the environment

    -> ReaderT r m a
    -- ^ Computation to run in the
    --   modified environment

    -> ReaderT r' m a
withReaderT f m =
  ReaderT $ runReaderT m . f
```

In the next chapter, we'll meet the `State` monad, with which we can not only read in a value but also provide a new one. This will permit monadic actions to change the value referenced in the context across monadic sequences—and not just within that monadic action. We'll demonstrate this later.

22.10 You tend to see ReaderT, not Reader

`Reader` rarely stands alone. Usually, it's one `Monad` in a *stack* of multiple types providing a `Monad` instance, such as with a web application that uses `Reader` to give you access to context about the HTTP request. When used in that fashion, it's a monad *transformer*, and we put a letter T after the type to indicate when we're using it as such, so you'll usually see `ReaderT` in production Haskell code, rather than `Reader`.

Further, a `Reader` of `Int` isn't all that useful or compelling. Usually, if you have a `Reader`, it's of a record of several (possibly many) values that you're getting out of the `Reader`.

22.11 Chapter exercises

A warm-up stretch

These exercises are designed to be a warm-up and to get you using some of the stuff we've learned in the last few chapters. While these exercises comprise code fragments from real code, they are simplified in order to be discrete exercises. That will allow us to highlight and practice some of the type manipulation from `Traversable` and `Reader`, both of which are tricky.

The first simplified part is that we're going to set up some toy data. In the real programs these are taken from, the data is coming from somewhere else—a database, for example. We just need some lists of numbers. We're going to use some functions from `Control.Applicative` and `Data.Maybe`, so we'll import those at the top of our practice file. We'll call our lists of toy data by common variable names, for simplicity:

```
module ReaderPractice where

import Control.Applicative
import Data.Maybe

x = [1, 2, 3]
y = [4, 5, 6]
z = [7, 8, 9]
```

The next thing we want to do is write some functions that zip those lists together and use `lookup` to find the value associated with a specified key in our zipped lists. For demonstration purposes, it's nice to have predictable outputs, so we recommend writing some that are concrete values, as well as one that can be applied to a variable:

```
lookup :: Eq a => a -> [(a, b)] -> Maybe b

-- zip x and y using 3 as the lookup key
xs :: Maybe Integer
xs = undefined
```

```
-- zip y and z using 6 as the lookup key
ys :: Maybe Integer
ys = undefined
```

It's also nice to have one that will return `Nothing`, like this one:

```
-- zip x and y using 4 as the lookup key
zs :: Maybe Integer
zs = lookup 4 $ zip x y

-- now zip x and z using a
-- variable lookup key
z' :: Integer -> Maybe Integer
z' n = undefined
```

Now, we want to add the ability to make a `Maybe (,)` of values using `Applicative`. Have `x1` make a tuple of `xs` and `ys` and `x2` make a tuple of of `ys` and `zs`. Also, write `x3`, which takes one input and makes a tuple of the results of two applications of `z'` from above:

```
x1 :: Maybe (Integer, Integer)
x1 = undefined

x2 :: Maybe (Integer, Integer)
x2 = undefined

x3 :: Integer
   -> (Maybe Integer, Maybe Integer)
x3 = undefined
```

Your outputs from those should look like this:

```
*ReaderPractice> x1
Just (6,9)
*ReaderPractice> x2
Nothing
*ReaderPractice> x3 3
(Just 9,Just 9)
```

Next, we're going to make some helper functions. Let's use `uncurry` to allow us to add the two values that are inside a tuple:

```
uncurry :: (a -> b -> c) -> (a, b) -> c
```

The first argument is a function. In this case, we want it to be addition. `summed` is uncurry with addition as the first argument:

```
summed :: Num c => (c, c) -> c
summed = undefined
```

And now, we'll make a function similar to some we've seen before that lifts a Boolean function over two partially applied functions:

```
bolt :: Integer -> Bool
-- use &&, >3, <8
bolt = undefined
```

Finally, we use `fromMaybe` in the `main` exercise, so let's look at that:

```
fromMaybe :: a -> Maybe a -> a
```

You give it a default value and a `Maybe` value. If the `Maybe` value is a `Just a`, it will return the `a` value. If the value is `Nothing`, it returns the default value instead:

```
*ReaderPractice> fromMaybe 0 xs
6
*ReaderPractice> fromMaybe 0 zs
0
```

Now, we'll cobble together a `main`, so that in one call, we can execute several things at once:

```
main :: IO ()

main = do
  print $
    sequenceA [Just 3, Just 2, Just 1]

  print $ sequenceA [x, y]
  print $ sequenceA [xs, ys]
```

```
  print $ summed <$> ((,) <$> xs <*> ys)
  print $ fmap summed ((,) <$> xs <*> zs)
  print $ bolt 7
  print $ fmap bolt z
```

When you run this in GHCi, your results should look like this:

```
*ReaderPractice> main
Just [3,2,1]
[[1,4],[1,5],[1,6],[2,4],[2,5],[2,6],[3,4],[3,5],[3,6]]
Just [6,9]
Just 15
Nothing
True
[True,False,False]
```

Next, we're going to add one that combines `sequenceA` and `Reader` in a somewhat surprising way (add this to `main`):

```
print $ sequenceA [(>3), (<8), even] 7
```

The type of `sequenceA` is:

```
sequenceA :: (Applicative f, Traversable t)
          => t (f a) -> f (t a)
-- so in this:
sequenceA [(>3), (<8), even] 7
-- f ~ (->) a and t ~ []
```

We have a `Reader` for the `Applicative` (functions) and a traversable for the list. Pretty handy. We're going to call this function `sequA` for the purposes of the following exercises:

```
sequA :: Integral a => a -> [Bool]
sequA m = sequenceA [(>3), (<8), even] m
```

And henceforth let:

```
summed <$> ((,) <$> xs <*> ys)
```

Be known as `s'`.

OK, your turn. Within the `main` above, write the following (you can delete everything after `do` now, if you prefer—just remember to use `print` to be able to print the results of what you're adding):

1. Fold the Boolean conjunction operator over the list of results of `sequA` (applied to some value).
2. Apply `sequA` to `s'`—you'll need `fromMaybe`.
3. Apply `bolt` to `ys`—you'll need `fromMaybe`.

Rewriting Shawty

Remember the URL shortener? Instead of manually passing the database connection `rConn` from `main` to the app function that generates a Scotty app, use `ReaderT` to make the database connection available. We know you haven't seen the transformer variant yet, and we'll explain transformers soon, but you should try to do the transformation mechanically. Use this version of the app: `https://github.com/bitemyapp/shawty-prime/blob/master/app/Main.hs`.

22.12 Definition

A *monad transformer* is a special type that takes a monad as an argument and returns a monad as a result. It allows us to combine two monads into one that shares the behaviors of both, such as allowing us to add exception handling to a `State` monad. It is somewhat common to build a *stack* of transformers that creates one large monad with features from several different monads. For example, we can roll `Reader`, `Either`, and `IO` together to get a monad that captures the behavior of waiting for an argument that will get passed around to multiple functions but is likely to come in via some kind of IO action and has a possibility of failure that we might like to catch. Often, this stack will be given a type alias for convenience.

22.13 Follow-up resources

1. Haskell Wiki. *All About Monads*. The Reader monad.
   `https://wiki.haskell.org/All_About_Monads#The_Reader_monad`

2. Bryan O'Sullivan, Don Stewart, and John Goerzen. *Real World Haskell*. Programming with monads. See the section on the reader monad.
`http://book.realworldhaskell.org/read/programming-with-monads.html`

Chapter 23

State

> Four centuries ago, Descartes pondered the mind-body problem: how can incorporeal minds interact with physical bodies? Today, computing scientists face their own version of the mind-body problem: how can virtual software interact with the real world?
>
> Philip Wadler

23.1 State

What if I need state? In Haskell, we have many means of representing, accessing, and modifying state. We can think of state as data that exists in addition to the inputs and outputs of our functions, data that can potentially change after each function is evaluated.

In this chapter, we will:

- Talk about what *state* means.
- Explore some ways of handling state in Haskell.
- Generate some more random numbers.
- Examine the `State` newtype and `Monad` instance.

23.2 What is state?

The concept of state originates in the circuit and automata theory that much of computer science and programming began with. The simplest form of state could be understood as a light switch. A light switch has two possible states, on or off. That disposition of the light switch, being on or off, could be understood as its *state*. Similarly, transistors in computers have binary states of being on or off. This is a very low-level way of seeing it, but it maps onto the *state* that exists in computer memory.

In most imperative programming languages, statefulness is pervasive, implicit, and not referenced in the types of your functions. In Haskell, we're not allowed to secretly change some value—all we can do is accept arguments and return a result. The `State` type in Haskell is a means of expressing state that may change in the course of evaluating code without resort to mutation. The monadic interface for `State` is, much as you've seen already, more of a convenience than a strict necessity for working with `State`.

We have the option to capture the idea and convenience of a value that potentially changes with each computation without resorting to mutability. `State` captures this idea and cleans up the bookkeeping required. If you need in-place mutation, then the `ST` type is what you want, and we mention that briefly in later chapters.

In Haskell, if we use the `State` type and its associated `Monad` (again, for convenience, as it's not strictly necessary), we can have state that:

1. Doesn't require `IO`.
2. Is limited only to the data in our `State` container.
3. Maintains referential transparency.
4. Is explicit in the types of our functions.

There are other means of sharing data within a program that are designed for different needs than the `State` datatype itself. `State` is appropriate when you want to express your program in terms of values that potentially vary with each evaluation step, which can be read and modified, but don't otherwise have specific operational constraints.

23.3 Random numbers

As we did in the previous chapter, we'll start with an extended example. This will help you get an idea of the problem we're trying to solve with the `State` datatype.

We'll be using the `random`[1] library, version 1.1, in this example.

First, let's get an overview of some of the functions we'll be using here. We used the `System.Random` library back in Chapter 13, where we built the hangman game, but we'll be using some different functions for this example. These are broad strokes; we aren't going to go into great detail about how these generators work.

`System.Random` is designed to generate pseudorandom values. You can generate those values by providing a seed value or by using the system-initialized generator. We'll be using the following from that library:

1. One of the types we'll be seeing here, `StdGen`, is a datatype that is a product of two `Int32` values. So, a value of type `StdGen` always comprises two `Int32` values. They are the seed values used to generate the next random number.
2. `mkStdGen` has the type:

```
mkStdGen :: Int -> StdGen
```

[1] `https://hackage.haskell.org/package/random`

We'll ignore the implementation at this point, because those details aren't important here. The idea is that it takes an `Int` argument and maps it onto a generator to return a value of type `StdGen`, which is a pair of `Int32` values.

3. `next` has the type:

```
next :: g -> (Int, g)
```

Where `g` is a value of type `StdGen`. The `Int` that is first in the tuple is the pseudorandom number generated from the `StdGen` value; the second value is a new `StdGen` value.

4. `random` has the type:

```
random :: (RandomGen g, Random a)
       => g -> (a, g)
```

This is similar to `next` but allows us to generate random values that aren't numbers. The range generated will be determined by the type.

Now, let's do a little demonstration of these functions:

```
Prelude> import System.Random
Prelude> mkStdGen 0
1 1
Prelude> :t mkStdGen 0
mkStdGen 0 :: StdGen
Prelude> sg = mkStdGen 0
Prelude> :t next sg
next sg :: (Int, StdGen)
Prelude> next sg
(2147482884,40014 40692)
Prelude> next sg
(2147482884,40014 40692)
```

We get the same answer twice, because the underlying function that decides which values are returned is pure—the type doesn't permit the performance of any effects. Define a new version of

`sg` that provides a different input value to `mkStdGen`, and see what happens.

So, we have a value called `next sg`. Now, if we want to use that to generate the next random number, we need to once more feed the `StdGen` value from that tuple to `next`. We can use `snd` to extract that `StdGen` value and pass it as the "next" input to `next`:

```
Prelude> snd (next sg)
40014 40692
Prelude> newSg = snd (next sg)
Prelude> :t newSg
newSg :: StdGen
Prelude> next newSg
(2092764894,1601120196 1655838864)
```

You'll keep getting the same results from `next`, but you can extract that `StdGen` value and pass it to `next` *again* to get a new tuple:

```
Prelude> next (snd (next newSg))
(1679949200,1635875901 2103410263)
```

Now, we'll look at a few examples using `random`. Because `random` can generate values of different types, we need to specify the type to use:

```
Prelude> :t random newSg
random newSg :: Random a => (a, StdGen)
Prelude> random newSg :: (Int, StdGen)
(138890298504988632,439883729 1872071452)
Prelude> random newSg :: (Double, StdGen)
(0.41992072972993366,439883729 1872071452)
```

Simple enough, but what if we want a number within a range?

```
Prelude> :t randomR
randomR :: (RandomGen g, Random a) =>
  (a, a) -> g -> (a, g)
Prelude> r = randomR (0, 3) newSg
Prelude> r :: (Int, StdGen)
(1,1601120196 1655838864)
Prelude> r :: (Double, StdGen)
(1.259762189189801,439883729 1872071452)
```

We have to pass the new state of the random number generator to the `next` function to get a new value:

```
Prelude> :{
Prelude| rx :: (Int, StdGen)
Prelude| rx = random (snd sg3)
Prelude| :}
Prelude> rx
(2387576047905147892,1038587761 535353314)
Prelude> snd rx
1038587761 535353314
```

This chaining of state can get tedious. Addressing this tedium is our aim in this chapter.

23.4 The State newtype

`State` is defined in a newtype, like `Reader` in the previous chapter, and that type looks like this:

```
newtype State s a =
  State { runState :: s -> (a, s) }
```

It's initially a bit strange looking, but you might notice some similarity to the `Reader` newtype:

```
newtype Reader r a =
  Reader { runReader :: r -> a }
```

A newtype has the same underlying representation as the type it wraps. This is because the newtype wrapper disappears at compile time. Therefore, the function contained in the newtype must be isomorphic to the type it wraps. That is, there must be a way to go from the newtype to the thing it wraps and back again without losing information. For example, the following code demonstrates an isomorphism:

```
type Iso a b = (a -> b, b -> a)

newtype Sum a = Sum { getSum :: a }

sumIsIsomorphicWithItsContents
  :: Iso a (Sum a)
sumIsIsomorphicWithItsContents =
  (Sum, getSum)
```

Whereas the following code does not, because it might not work:

```
(a -> Maybe b, b -> Maybe a)
```

And the example below is not an isomorphism for two reasons. First, you lose information whenever there is more than one element in `[a]`. Second, `[a] -> a` is a partial function, because there might not be any elements in the list:

```
[a] -> a, a -> [a]
```

With that in mind, let us look at the `State` data constructor and `runState` record accessor as our means of putting a value in and taking a value out of the `State` type:

```
State :: (s -> (a, s)) -> State s a

runState :: State s a -> s -> (a, s)
```

`State` is a function that takes an input state and returns an output value, `a`, tupled with the new state value. The key is that the previous state value from each application is chained to the next one, and this is not an uncommon pattern. `State` is often used for things like random number generators, solvers, games, and carrying working memory while traversing a data structure. The polymorphism means you don't have to make a new state for each possible instantiation of `s` and `a`.

Let's get back to our random numbers, noting that `random` looks an awful lot like `State`:

```
random :: (Random a)
       => StdGen -> (a, StdGen)
State { runState
            :: s -> (a, s)    }
```

If we look at the type of `randomR`, once partially applied, it should also remind you of `State`:

```
randomR :: (...) => (a, a) -> g -> (a, g)
State {    runState ::        s -> (a, s) }
```

23.5 Throw down

Let's use this kit to generate dice, such as for a game:

```
module RandomExample where

import System.Random

-- Six-sided die
data Die =
    DieOne
  | DieTwo
  | DieThree
  | DieFour
  | DieFive
  | DieSix
  deriving (Eq, Show)
```

As you might expect, we'll be using the `random` library and a simple `Die` datatype to represent a six-sided die:

```
intToDie :: Int -> Die
intToDie n =
  case n of
    1 -> DieOne
    2 -> DieTwo
    3 -> DieThree
    4 -> DieFour
    5 -> DieFive
    6 -> DieSix

    -- Use error sparingly
    x ->
      error $
        "intToDie got non 1-6 integer: "
        ++ show x
```

Don't use `error` outside of experiments like this, or in cases where the branch you're ignoring is provably impossible. We do not use the word *provably* here lightly.[2]

Now we need to roll the dice:

```
rollDieThreeTimes :: (Die, Die, Die)
rollDieThreeTimes = do
  let s = mkStdGen 0
      (d1, s1) = randomR (1, 6) s
      (d2, s2) = randomR (1, 6) s1
      (d3, _)  = randomR (1, 6) s2
  (intToDie d1, intToDie d2, intToDie d3)
```

This code isn't optimal, but it does work. It will produce the same results every time, because it is free of effects, but you can make it produce a new result on a new roll of the dice if you modify the start value. Try it a couple of times to see what we mean. It seems unlikely that this will develop into a gambling addiction, but in the event it does, the authors disclaim any liability.

[2]Because partial functions are a pain, you should only use an error like this when the branch that would spawn the error can literally never happen. Unexpected software failures are often due to things like this. It is also completely unnecessary in Haskell; we have good alternatives, like using `Maybe` or `Either`. The only reason we didn't here is to keep it simple and focus attention on the `State Monad`.

So, how exactly can we improve the suboptimal code in our `rollDieThreeTimes` function above? Well, with `State`, of course!

```
module RandomExample2 where

import Control.Applicative (liftA3)
import Control.Monad (replicateM)
import Control.Monad.Trans.State
import System.Random
import RandomExample
```

First, we'll add some new imports. You'll need `transformers` to be installed for the `State` import to work, but that should have come with your GHC installation, so you should be good to go.

By using `State`, we can factor out the generation of a single `Die`:

```
rollDie :: State StdGen Die
rollDie = state $ do
  (n, s) <- randomR (1, 6)
  return (intToDie n, s)
```

For our purposes, the `state` function is a constructor that takes a `State`-like function and embeds it in the `State` monad transformer. Ignore the transformer part for now—we'll get there. The `state` function has the following type:

```
state :: Monad m
      => (s -> (a, s))
      -> StateT s m a
```

Note that we're binding the result of `randomR` out of the `State` monad the `do` block is in rather than using `let`. This is still more verbose than is necessary. We can lift our `intToDie` function over the `State`:

```
rollDie' :: State StdGen Die
rollDie' =
  intToDie <$> state (randomR (1, 6))
```

`State StdGen` has a final type argument of `Int`. We lift `Int -> Die` over it and transform that final type argument to `Die`. We'll exercise more brevity upfront, in the next function:

```
rollDieThreeTimes'
  :: State StdGen (Die, Die, Die)
rollDieThreeTimes' =
  liftA3 (,,) rollDie rollDie rollDie
```

Here, we lift the 3-tuple constructor over three `State` actions that produce `Die` values when given an initial state to work with. How does this work, in practice?

```
Prelude> evalState rollDieThreeTimes' (mkStdGen 0)
(DieSix,DieSix,DieFour)
Prelude> evalState rollDieThreeTimes' (mkStdGen 1)
(DieSix,DieFive,DieTwo)
```

Seems to work fine. Again, the same inputs give us the same results. What if we want a list of `Die` instead of a tuple?

```
-- Seems appropriate?
repeat :: a -> [a]

infiniteDie :: State StdGen [Die]
infiniteDie = repeat <$> rollDie
```

Does this `infiniteDie` function do what we want or expect? What is it repeating?

```
Prelude> gen = (mkStdGen 0)
Prelude> take 6 $ evalState infiniteDie gen
[DieSix,DieSix,DieSix,DieSix,DieSix,DieSix]
```

We already know based on previous inputs that the first three values shouldn't be identical for a seed value of 0. So what is happening? What's happening is we repeat a single *die value*—we don't repeat the state action that *produces a die*. This is what we need:

```
replicateM :: Monad m
           => Int -> m a -> m [a]

nDie :: Int -> State StdGen [Die]
nDie n = replicateM n rollDie
```

And when we use it?

```
Prelude> evalState (nDie 5) (mkStdGen 0)
[DieSix,DieSix,DieFour,DieOne,DieFive]
Prelude> evalState (nDie 5) (mkStdGen 1)
[DieSix,DieFive,DieTwo,DieSix,DieFive]
```

We get precisely what we want.

Keep on rolling

In the following example, we keep rolling a single die until we reach or exceed a sum of 20:

```
rollsToGetTwenty :: StdGen -> Int
rollsToGetTwenty g = go 0 0 g

  where
    go :: Int -> Int -> StdGen -> Int
    go sum count gen
      | sum >= 20 = count
      | otherwise =

        let (die, nextGen) =
              randomR (1, 6) gen
        in go (sum + die)
              (count + 1) nextGen
```

Let's see it in action:

```
Prelude> rollsToGetTwenty (mkStdGen 0)
5
Prelude> rollsToGetTwenty (mkStdGen 0)
5
```

We can also use `randomIO`, which uses `IO` to get a new value each time, without needing to create a unique value for the `StdGen`:

```
Prelude> :t randomIO
randomIO :: Random a => IO a
Prelude> rs = (rollsToGetTwenty . mkStdGen)
```

```
Prelude> rs <$> randomIO
6
Prelude> rs <$> randomIO
7
```

Under the hood, it's the same interface and `State` `Monad`-driven mechanism, but it's mutating a single, globally-used `StdGen` to walk the generator forward on each use. See the `random` library source code to see how this works.

Exercises: Roll your own

1. Refactor `rollsToGetTwenty` so that the limit is an argument to the function:

```
rollsToGetN :: Int -> StdGen -> Int
rollsToGetN = undefined
```

2. Change `rollsToGetN` to record the series of dice that are rolled, in addition to the count of the total number of rolls:

```
rollsCountLogged :: Int
                 -> StdGen
                 -> (Int, [Die])
rollsCountLogged = undefined
```

23.6 Write State for yourself

Use the datatype definition from the beginning of this chapter, with the name changed to avoid conflicts, in case you have `State` imported from `transformers` or `mtl`. We're calling it `Moi`, because we enjoy allusions to famous quotations[3]—feel free to change the name if you wish to protest absolute monarchy, but change them consistently throughout:

```
newtype Moi s a =
  Moi { runMoi :: s -> (a, s) }
```

[3]We are referring to the (possibly apocryphal) quotation attributed to the French king, Louis XIV: "L'etat, c'est moi." For those of you who do not speak French, it means, "I am the state." Cheers.

State Functor

Implement the `Functor` instance for `State`:

```
instance Functor (Moi s) where
  fmap :: (a -> b) -> Moi s a -> Moi s b
  fmap f (Moi g) = ???
```

```
Prelude> f = (+1) <$> (Moi $ \s -> (0, s))
Prelude> runMoi f 0
(1,0)
```

State Applicative

Write the `Applicative` instance for `State`:

```
instance Applicative (Moi s) where
  pure :: a -> Moi s a
  pure a = ???

  (<*>) :: Moi s (a -> b)
        -> Moi s a
       -> Moi s b
  (Moi f) <*> (Moi g) =
    ???
```

State Monad

Write the `Monad` instance for `State`:

```
instance Monad (Moi s) where
  return = pure

  (>>=) :: Moi s a
        -> (a -> Moi s b)
        -> Moi s b
  (Moi f) >>= g =
    ???
```

23.7 Get a coding job with one weird trick

Some companies will use the FizzBuzz[4] problem to screen (not so much *test*) candidates applying to software positions. The instructions are as follows:

> Write a program that prints the numbers from 1 to 100. But for multiples of three print `"Fizz"` instead of the number and for the multiples of five print `"Buzz"`. For numbers which are multiples of both three and five print `"FizzBuzz"`.

A typical FizzBuzz solution in Haskell looks something like:

```
fizzBuzz :: Integer -> String
fizzBuzz n | n `mod` 15 == 0 = "FizzBuzz"
           | n `mod` 5  == 0 = "Buzz"
           | n `mod` 3  == 0 = "Fizz"
           | otherwise       = show n

main :: IO ()
main =
  mapM_ (putStrLn . fizzBuzz) [1..100]
```

A FizzBuzz using `State` is a suitable punishment for asking a software candidate to write this in person after presumably getting through a couple phone screens. Let's look at what a version with `State` might look like:

```
import Control.Monad
import Control.Monad.Trans.State

fizzBuzz :: Integer -> String
fizzBuzz n | n `mod` 15 == 0 = "FizzBuzz"
           | n `mod` 5  == 0 = "Buzz"
           | n `mod` 3  == 0 = "Fizz"
           | otherwise       = show n

fizzbuzzList :: [Integer] -> [String]
fizzbuzzList list =
  execState (mapM_ addResult list) []
```

[4] http://c2.com/cgi/wiki?FizzBuzzTest

```
addResult :: Integer -> State [String] ()
addResult n = do
    xs <- get
    let result = fizzBuzz n
    put (result : xs)

main :: IO ()
main =
  mapM_ putStrLn $
    reverse $ fizzbuzzList [1..100]
```

Note that `State` is a type alias of the `StateT` we import.

The good part here is that we're collecting data initially before dumping the results to standard output via `putStrLn`. The bad is that we're reversing a list. Reversing singly-linked lists is not great, even in Haskell, and won't terminate on an infinite list. One of the issues is that we're accepting an input that defines the numbers we'll use FizzBuzz on linearly, from beginning to end.

There are a couple of ways we could handle this. One is to use a data structure with cheaper appending to the end. Using `++` recursively can be very slow, so let's use something that can append in constant time. The counterpart to `[]` that has this property is the difference list,[5] which provides an operation that can append in constant time, that is, the time it takes doesn't grow with the length of the list. Programmers sometimes refer to this as O(1) time, using so-called Big O notation. Let's see it in action:

```
import Control.Monad
import Control.Monad.Trans.State
import qualified Data.DList as DL

fizzBuzz :: Integer -> String
fizzBuzz n | n `mod` 15 == 0 = "FizzBuzz"
           | n `mod` 5  == 0 = "Buzz"
           | n `mod` 3  == 0 = "Fizz"
           | otherwise       = show n
```

[5] https://github.com/spl/dlist

```
fizzbuzzList :: [Integer] -> [String]
fizzbuzzList list =
  let dlist =
        execState (mapM_ addResult list)
                  DL.empty
     -- convert back to normal list
  in DL.apply dlist []

addResult :: Integer
          -> State (DL.DList String) ()
addResult n = do
    xs <- get
    let result = fizzBuzz n
    -- snoc appends to the end, unlike
    -- cons which adds to the front
    put (DL.snoc xs result)

main :: IO ()
main =
  mapM_ putStrLn $ fizzbuzzList [1..100]
```

We can clean this up further. If you have GHC 7.10 or newer, `mapM_` will specify a `Foldable` type, not only a list:

```
Prelude> :t mapM_
mapM_ :: (Monad m, Foldable t) =>
  (a -> m b) -> t a -> m ()
```

By letting `DList`'s `Foldable` instance do the conversion to a list for us, we can eliminate some code:

```
fizzbuzzList :: [Integer]
             -> DL.DList String
fizzbuzzList list =
  execState (mapM_ addResult list) DL.empty
```

```
addResult :: Integer
          -> State (DL.DList String) ()
addResult n = do
    xs <- get
    let result = fizzBuzz n
    put (DL.snoc xs result)

main :: IO ()
main =
  mapM_ putStrLn $ fizzbuzzList [1..100]
```

`DList`'s `Foldable` instance converts to a list before folding because of limitations specific to the datatype. You get cheap appending, but you give up the ability to "see" what you've built unless you're willing to do all the work of building the structure. We'll discuss this in more detail in a forthcoming chapter.

One thing that may strike you here is that the use of `State` is superfluous. That's good! It's not common that you *need* `State`, as such, in Haskell. You might use a different form of `State` called `ST` as a selective optimization, but `State` itself is a stylistic choice that falls out of what the code is telling you. Don't feel compelled to use or not use `State`. Please frighten some interviewers with a spooky FizzBuzz. Make something even weirder than what we've shown you here!

FizzBuzz differently

It's an exercise! Rather than changing the underlying data structure, fix our reversing FizzBuzz by changing the code in the following way:

```
fizzbuzzFromTo :: Integer
               -> Integer
               -> [String]
fizzbuzzFromTo = undefined
```

Continue to use consing in the construction of the result list, but have it come out in the right order to begin with by *enumerating the sequence backward*. This sort of tactic is more commonly how you'll want to fix your code when you're quashing unnecessary reversals.

23.8 Chapter exercises

Write the following functions. You'll want to use your own `State` type for which you've defined `Functor`, `Applicative`, and `Monad` instances.

1. Construct a `State` where the state is also the value you return:

```
get :: State s s
get = ???
```

Expected output:

```
Prelude> runState get "curryIsAmaze"
("curryIsAmaze","curryIsAmaze")
```

2. Construct a `State` where the resulting state is the argument provided, and the value defaults to unit:

```
put :: s -> State s ()
put s = ???

Prelude> runState (put "blah") "woot"
((),"blah")
```

3. Run the `State` with `s` and get the state that results:

```
exec :: State s a -> s -> s
exec (State sa) s = ???

Prelude> exec (put "wilma") "daphne"
"wilma"
Prelude> exec get "scooby papu"
"scooby papu"
```

4. Run the `State` with `s` and get the value that results:

```
eval :: State s a -> s -> a
eval (State sa) = ???
```

```
Prelude> eval get "bunnicula"
"bunnicula"
Prelude> eval get "stake a bunny"
"stake a bunny"
```

5. Write a function that applies a function to create a new `State`:

```
modify :: (s -> s) -> State s ()
modify = undefined
```

It should behave like the following:

```
Prelude> f = modify (+1)
Prelude> runState f 0
((),1)
Prelude> runState f >> f 0
((),2)
```

You don't need to compose them—you can throw away the result, because it returns unit for `a`, anyway.

23.9 Follow-up resources

1. Haskell Wiki. *State Monad*.
   `https://wiki.haskell.org/State_Monad`

Chapter 24

Parser Combinators

> Within a computer, natural language is unnatural.
>
> Alan Perlis

24.1 Parser combinators

The word *parse* comes from the Latin word for *parts* and means to analyze a sentence and label the syntactic role, or part of speech, of each component. Language teachers once emphasized this ability, because it forced students to think about the structure of sentences, the relationships among the parts, and the connection between the structure and the meaning of the whole. Diagramming sentences was common, because it made parsing visual and somewhat concrete.

It is now common to represent grammatical structures of natural languages as trees, so that a sentence such as:

```
A boy plays with the dog.
```

Might be thought to have an underlying representation, such as:

```
          S(entence)
        /   \
    A boy   plays (verb)
(subject)       \
                with (preposition)
                   \
                   the dog (object)
```

We are not here to become linguists, but parsing in computer science is related to the parsing of natural language sentences. The core idea of parsing in programming is to accept serialized input in the form of a sequence of characters (textual data) or bytes (raw binary data) and turn that into a value of a structured datatype. Serialized data is data that has been translated into a format, such as JSON or XML,[1] that can be stored or transmitted across a network connection. Parsing breaks up that chunk of data and allows you to find and process the parts you care about.

If we write a computer program to parse a sentence into a *very* simplified model of English grammar, it could look something like the tree above. Often, when we are parsing things, the structured datatype that results will look something like a tree. In Haskell, we

[1] If you do not know what JSON and XML are yet, try not to get too hung up on that. All that matters at this point is that they are standard data formats. We'll look at JSON in more detail later in the chapter.

can sometimes end up having a tree data structure, because recursive types are so easy to express in Haskell.

In this chapter, we will:

- Use a parsing library to cover the basics of parsing.
- Demonstrate the awesome power of parser combinators.
- Marshall and unmarshall some JSON data.
- Talk about tokenization.

24.2 A few more words of introduction

In this chapter, we will not look too deeply into the types of the parsing libraries we're using, learn about every sort of parser there is, or artisanally handcraft all of our parsing functions ourselves.

These are thoroughly considered decisions. Parsing is a *huge* field of research in its own right with connections that span natural language processing, linguistics, and programming language theory. This topic could easily fill a book in itself (in fact, it has). The underlying types and type classes of the libraries we'll be using are complicated. To be sure, if you enjoy parsing and expect to do it a lot, those are things you'd want to learn; they are simply out of the scope of this book.

This chapter takes a different approach than previous chapters. The focus is on enabling you to use Haskell's parsing libraries—not to be a master of parsing and writing parsers, in general. This is not the bottom-up approach you may be accustomed to. By necessity, we're working outside-in and trying to cover what you're likely to *need*. Depending on your specific interests, you may find this chapter too long or not nearly long enough.

24.3 Understanding the parsing process

A *parser* is a function that takes some textual input (it could be a `String` or another datatype, such as `ByteString` or `Text`) and returns some *structure* as an output. That structure might be a tree, for example, or an indexed map of locations in the parsed data. Parsers analyze structure in conformance with rules specified in a grammar, whether

it's a grammar of a human language, a programming language, or a format such as JSON.

A *parser combinator* is a higher-order function that takes parsers as input and returns a new parser as output. You may remember our brief discussion of combinators way back in the lambda calculus chapter. Combinators are expressions with no free variables.

The standard for what constitutes a combinator with respect to parser combinators is a little looser. Parsers are functions, so parser combinators are higher-order functions that can take parsers as arguments. Usually, the argument passing is elided, because the interface of parsers will often be like the `State` monad, which permits `Reader`-style implicit argument passing. Among other things, combinators allow for recursion and for gluing together parsers in a modular fashion to parse data according to complex rules.

For computers, parsing is something like reading when you're really young. Perhaps you were taught to trace the letters with your finger for phonetic pronunciation. Later, you were able to follow word by word, and then you started scanning with your eyes. Eventually, you learned how to read with subvocalization.

Since we didn't use an analogy for Monad

We're going to run through some code now that will demonstrate the idea of parsing. Let's begin by installing the parsing library `trifecta`,[2] then work through a short demonstration of what it does. We'll talk more about the design of `trifecta` in a while. For now, we're going to use it in a state of somewhat ignorant bliss.

Let's put up some code:

```
module LearnParsers where

import Text.Trifecta

stop :: Parser a
stop = unexpected "stop"
```

[2]We'll be using this version of the `trifecta` library for the example code in this chapter: `http://hackage.haskell.org/package/trifecta-1.5.2`.

`unexpected` is a means of throwing errors in parsers like `trifecta`, which are an instance of the `Parsing` type class. Here, we're using it to make the parser fail for demonstration purposes.

What demonstration purposes?

We're glad you asked! The basic idea behind a parser is that you're moving a sort of cursor around a linear stream of text. It's simplest to think of the individual units within the stream as characters or ideographs, though you'll want to start thinking of your parsing problems in *chunkier* terms as you progress. The idea is that this cursor is a bit like you're reading the text with your finger:

```
Julie bit Papuchon
^
```

Then, let us say we parsed the word "Julie"—we've now consumed that input, so the cursor will be at "bit":

```
Julie bit Papuchon
      ^
```

If we weren't expecting the word "bit," our parser could fail here, and we'd get an error at the word "bit" like that. However, if we do parse the word "bit" successfully, and thus consume that input, it might look something like this:

```
Julie bit Papuchon
          ^
```

The analogy we're using here isn't perfect. One of the hardest problems in writing parsers, especially the parser libraries themselves, is making it easy to express things the way the programmer would like but still have the resulting parser be *fast*.

Back to the code

With the cursor analogy in mind, let's return to the module we started.

We'll first make a little function that only parses one character and then sequence that with `stop` to make it read that one character and then die:

```
-- read a single character '1'
one = char '1'

-- read a single character '1', then die
one' = one >> stop
-- equivalent to char '1' >> stop
```

For `one'`, we're using the sequencing operator from `Monad` to combine two parsers, `stop` and `char '1'`. Given the type of `>>`:

```
(>>) :: Monad m => m a -> m b -> m b
```

It's safe to assume that whatever `char '1'` returns in the following expression gets thrown away:

```
char '1' >> stop
```

Critically, any *effect* the `m a` action has on the monadic context remains. The result value of the parse function gets thrown away, but the effect of "moving the cursor" remains. Another possible effect is causing the parse to fail.

A bit like...

`State`. Plus failure. No, seriously, take a look at this definition of the `Parser` type:

```
type Parser a = String -> Maybe (a, String)
```

You can read this as:

1. Await a string value.
2. Produce a result that may or may not succeed (a `Nothing` value means the parse failed).
3. Return a tuple of the value you want and whatever's left of the string that you didn't consume to produce a value of type `a`.

Then, remind yourself of what `Reader` and `State` look like:

```
newtype Reader r a =
  Reader { runReader :: r -> a }

newtype State s a =
  State { runState :: s -> (a, s) }
```

If you have convinced yourself that `State` is an elaboration of `Reader` and that you can see how the `Parser` type looks sorta like `State`, we can move on.

The idea here with the `Parser` type is that the `State` is handling the fact that you need to await an eventual text input and that having parsed something out of that text input results in a new state of the input stream. It also lets you return a value independent of the state, while `Maybe` handles the possibility of the parser failure.

If we were to look at the underlying pattern of a parsing function such as `char`, you can see the `State`-ish pattern. Please understand that while this should work as a character-parsing function, we are simplifying here, and this is not what the source code of any modern parsing library will look like:

```
-- rudimentary char
-- demo only, this won't work as is.
char :: Char -> Parser Char
char c =
  Parser $ \s ->
    case s of
      (x:xs) -> if c == x
                then [(c, xs)]
                else []
      _ -> []
```

We could encode the possibility of failure in that by adding a `Maybe`, but at this point, it isn't important, because we're using a library that has encoded the possibility of failure for us. It has also optimized the heck out of `char` for us. But we want to show you how the underlying function is the `\s ->` embedded in the `Parser` data constructor.

Consider the type of a Hutton-Meijer parser:

```
-- from Text.ParserCombinators.HuttonMeijer
-- polyparse-1.11

type Token = Char
newtype Parser a =
  P ([Token] -> [(a, [Token])])

-- Same thing, differently formatted
type Parser' a = String -> [(a, String)]
```

This changes things from the previous, less common but simpler variant, by allowing you to express a range of possibly valid parses starting from the input provided. This is more powerful than the `Maybe` variant, but this design isn't used in popular Haskell parser combinator libraries any longer. Although the underlying implementation has changed dramatically with new discoveries and designs, most parsing libraries in Haskell are going to have an interface that behaves a bit like `State`, in that the act of parsing things has an observable effect on one or more bits of state.

If we were talking about `State`, this means any `put` to the `State` value would be observable to the next action in the same `Monad` (you can verify what follows in your REPL by importing `Control.Monad.Trans.State`). These examples use the transformer variant of `State`, but if you ignore the `T`, you should be able to get the basic idea:

```
get :: Monad m => StateT s m s
put :: Monad m => s -> StateT s m ()
runStateT :: StateT s m a -> s -> m (a, s)

Prelude> runStateT (put 8) 7
((),8)
Prelude> runStateT get 8
(8,8)
Prelude> runStateT (put 1 >> get) 8
(1,1)
Prelude> (runStateT $ put 1 >> get) 0
(1,1)
Prelude> rs = runStateT
Prelude> n = 10021490234890
```

```
Prelude> (rs $ put 2 >> get) n
(2, 2)
Prelude> (rs $ put 2 >> return 9001) 0
(9001,2)
```

Now, `put` returns a unit value, a throwaway value, so we're only evaluating it *for effect* anyway. It modifies the state but doesn't have any value of its own. So, when we throw away its value, we're left with its effect on the state, although `get` puts that value into both the `a` and `s` slots in the tuple.

This is an awful lot like what happens when we sequence a parsing function such as `char` with `stop`, as above. There is no real result of `char`, but it does change the state. The state here is the location of the cursor in the input stream. In reality, a modern and mature parser design in Haskell will often look about as familiar to you as the alien hellscape underneath the frozen crust of one of the moons of Jupiter. Don't take the idea of there being an actual cursor too literally, but there may be some utility in imagining it this way.

Back to our regularly scheduled coding

Onward with the code:

```
-- read two characters, '1' and '2'
oneTwo = char '1' >> char '2'

-- read two characters,
-- '1' and '2', then die
oneTwo' = oneTwo >> stop

testParse :: Parser Char -> IO ()
testParse p =
  print $ parseString p mempty "123"
```

The `p` argument is a parser. Specifically, it's a character parser. The functions `one` and `oneTwo` have the type `Parser Char`. You can check the types of `one'` and `oneTwo'` yourself.

We need to declare the type of `testParse` in order to `Show` what we parse because of ambiguity.

The key thing to realize here is that we're using parsers like values and combining them using the same stuff we use with ordinary functions or operators from the `Applicative` and `Monad` type classes. The structure that makes up the `Applicative` or `Monad` in this case is the `Parser` itself.

Next, we'll write a function to print a string to standard output (`stdout`) with a newline prefixed and then use that function as part of a `main` that will show us what we've got so far:

```
pNL s =
  putStrLn ('\n' : s)

main = do
  pNL "stop:"
  testParse stop

  pNL "one:"
  testParse one

  pNL "one':"
  testParse one'

  pNL "oneTwo:"
  testParse oneTwo

  pNL "oneTwo':"
  testParse oneTwo'
```

Let's run it and interpret the results. Since it's text on a computer screen instead of tea leaves, we'll call it science. If you remain unconvinced, you have our permission to don a white lab coat and print the output using a dot matrix printer.

Run `main`, and see what happens:

```
Prelude> main

stop:
Failure (interactive):1:1: error: unexpected
    stop
123<EOF>
^
```

We fail immediately before consuming any input in the above, so the caret in the error is at the beginning of our string value.

Next result:

```
one:
Success '1'
```

We parse a single character, the digit 1. The result is knowing we succeeded. But what about the rest of the input stream? Well, the thing we used to run the parser drops the rest of the input on the floor. There are ways to change this behavior, which we'll explain in the exercises.

Next up:

```
one':
Failure (interactive):1:2: error: unexpected
    stop
123<EOF>
 ^
```

We parse a single character successfully, then drop it, because we use `>>` to sequence it with `stop`. This means the cursor ends up one character forward due to the previous parser succeeding. Helpfully, `trifecta` tells us where our parser fails.

And for our last result:

```
oneTwo:
Success '2'

oneTwo':
Failure (interactive):1:3: error: unexpected
    stop
123<EOF>
  ^
```

It's the same as before, but we parse two characters individually. What if we we don't want to discard the first character we parse and instead parse "12"? See the exercises below!

Exercises: Parsing practice

1. There's a combinator that'll let us mark that we expect an input stream to be finished at a particular point in our parser. In the `parsers` library, this is simply called `eof` (end-of-file) and is in the `Text.Parser.Combinators` module. See if you can make the `one` and `oneTwo` parsers fail, because they don't exhaust the input stream.

2. Use `string` to make a `Parser` that parses "1", "12", and "123" out of the example input, respectively. Try combining it with `stop`, too. That is, a single parser should be able to parse all three of those strings. An example:

   ```
   Prelude> p123 "1"
   Success 1
   Prelude> p123 "12"
   Success 12
   Prelude> p123 "123"
   Success 123
   ```

   You can be more creative than this with the parser, if you want.

3. Try writing a `Parser` that does what `string` does, but using `char`.

Intermission: Parsing free jazz

Let us play with these parsers! We typically use the `parseString` function to run parsers, but if you figure out some other way that works for you, so be it! Here's some parsing free jazz, if you will, meant only to help develop your intuition about what's going on:

```
Prelude> import Text.Trifecta
Prelude> :t char
char :: CharParsing m => Char -> m Char
Prelude> :t parseString
parseString
  :: Parser a
  -> Text.Trifecta.Delta.Delta
  -> String
  -> Result a
Prelude> gimmeA = char 'a'
```

```
Prelude> :t parseString gimmeA mempty
parseString gimmeA mempty :: String -> Result Char

Prelude> parseString gimmeA mempty "a"
Success 'a'
Prelude> parseString gimmeA mempty "b"
Failure (interactive):1:1: error: expected: "a"
b<EOF>
^

Prelude> parseString (char 'b') mempty "b"
Success 'b'
Prelude> bToC = char 'b' >> char 'c'
Prelude> parseString bToC mempty "b"
Failure (interactive):1:2: error: unexpected
    EOF, expected: "c"
b<EOF>
 ^
Prelude> parseString btoC mempty "bc"
Success 'c'
Prelude> parseString bToC mempty "abc"
Failure (interactive):1:1: error: expected: "b"
abc<EOF>
^
```

Seems like we ought to have a way to say, "parse this string," rather than having to sequence the parsers of individual characters bit by bit, right? Turns out, we do:

```
Prelude> parseString (string "abc") mempty "abc"
Success "abc"
Prelude> parseString (string "abc") mempty "bc"
Failure (interactive):1:1: error: expected: "abc"
bc<EOF>
^
Prelude> parseString (string "abc") mempty "ab"
Failure (interactive):1:1: error: expected: "abc"
ab<EOF>
^
```

Importantly, it's not a given that a single parser exhausts all of its input—they only consume as much text as they need to produce the value of the type requested.

```
Prelude> ps = parseString
Prelude> ps (char 'a') mempty "abcdef"
Success 'a'
Prelude> stop = unexpected "stop pls"
Prelude> ps (char 'a' >> stop) mempty "abcdef"
Failure (interactive):1:2: error: unexpected
    stop pls
abcdef<EOF>
 ^
Prelude> ps (string "abc") mempty "abcdef"
Success "abc"
Prelude> ps (string "abc" >> stop) mempty "abcdef"
Failure (interactive):1:4: error: unexpected
    stop pls
abcdef<EOF>
   ^
```

Note that we can also parse UTF-8 encoded `ByteStrings` with `trifecta`:

```
Prelude> import Text.Trifecta
Prelude> :t parseByteString
parseByteString
  :: Parser a
     -> Text.Trifecta.Delta.Delta
     -> Data.ByteString.Internal.ByteString
     -> Result a
Prelude> parseByteString (char 'a') mempty "a"
Success 'a'
```

This ends the free jazz session. We now return to serious matters.

24.4 Parsing fractions

Now that we have some idea of what parsing is, what parser combinators are, and what the monadic underpinnings of parsing look like,

let's move on to parsing fractions. The top of this module should look like this:

```
{-# LANGUAGE OverloadedStrings #-}

module Text.Fractions where

import Control.Applicative
import Data.Ratio ((%))
import Text.Trifecta
```

We name the module `Text.Fractions`, because we're parsing *fractions* out of *text input*, and there's no need to be more clever about it than that. We're going to be using `String` inputs with `trifecta` at first, but you'll see why we threw an `OverloadedStrings` extension in there, later.

Now, on to parsing fractions! We'll start with some test inputs:

```
badFraction = "1/0"
alsoBad = "10"
shouldWork = "1/2"
shouldAlsoWork = "2/1"
```

Then, we'll write our actual parser:

```
parseFraction :: Parser Rational
parseFraction = do
  numerator <- decimal
--  [2]         [1]
  char '/'
-- [3]
  denominator <- decimal
--         [  4  ]
  return (numerator % denominator)
--   [5]          [6]
```

1. `decimal :: Integral a => Parser a`

   This is the type of `decimal` within the context of those functions. If you use GHCi to query the type of `decimal`, you will see a more polymorphic type signature.

2. Here `numerator` has the type `Integral a => a`.

3. `char :: Char -> Parser Char`

   As with `decimal`, if you query the type of `char` in GHCi, you'll see a more polymorphic type, but this is the type of `char`, in context.

4. Same deal as `numerator`, but when we match an integral number, we're binding the result to the name `denominator`.

5. The final result has to be a parser, so we embed our integral value in the `Parser` type by using `return`.

6. We construct ratios using the `%` infix operator:

   ```
   (%) :: Integral a
       => a -> a -> GHC.Real.Ratio a
   ```

   Then, the fact that our final result is a `Rational` makes the `Integral a => a` values into concrete `Integer` values.

   ```
   type Rational = GHC.Real.Ratio Integer
   ```

We'll put together a quick shim main function to run the parser against the test inputs and see the results:

```
main :: IO ()
main = do

  let parseFraction' =
        parseString parseFraction mempty

  print $ parseFraction' shouldWork
  print $ parseFraction' shouldAlsoWork

  print $ parseFraction' alsoBad
  print $ parseFraction' badFraction
```

Try not to worry about the `mempty` values—they might give you a clue about what's going on in `trifecta` under the hood, but it's not something we're going to explore in this chapter.

We will briefly note the type of `parseString`, which is how we're running the parser we created:

```
parseString :: Parser a
            -> Text.Trifecta.Delta.Delta
            -> String
            -> Result a
```

The first argument is the parser we're going to run against the input, the second is a `Delta`, the third is the `String` we're parsing, and then the final result is either the thing we want of type `a` or an error string to let us know that something went wrong. You can ignore the `Delta` thing—use `mempty` to provide the do-nothing input. We won't be covering deltas in this book, so consider it extra credit if you get curious.

Anyway, when we run the code, the results look like this:

```
Prelude> main
Success (1 % 2)
Success (2 % 1)
Failure (interactive):1:3: error: unexpected
    EOF, expected: "/", digit
10<EOF>
  ^
Success *** Exception: Ratio has zero denominator
```

The first two succeed properly. The third fails, because it can't parse a fraction out of the text `"10"`. The error is telling us that it ran out of text in the input stream while still waiting for the character `'/'`. The final error does not result from the process of parsing; we know that, because it is a `Success` data constructor. The final error instead results from trying to construct a ratio with a denominator that is zero—which makes no sense. We can reproduce the issue in GHCi:

```
Prelude> 1 % 0
*** Exception: Ratio has zero denominator
```

So, the parser result is tantamount to:

```
Prelude> Success (1 % 0)
Success *** Exception: Ratio has zero denominator
```

This is a problem, because exceptions *end* our programs. Observe:

```
main :: IO ()
main = do

  let parseFraction' =
        parseString parseFraction mempty

  print $ parseFraction' badFraction
  print $ parseFraction' shouldWork

  print $ parseFraction' shouldAlsoWork
  print $ parseFraction' alsoBad
```

We put the expression that throws an exception in the first line this time, and when we run it we get:

```
Prelude> main
Success *** Exception: Ratio has zero denominator
```

So, our program halts on the error. This is not great. You may be tempted to "handle" the error. Catching exceptions is OK, but this is a particular class of exceptions that means something is wrong with your program. You should eliminate the possibility of exceptions occurring in your programs where possible.

We'll talk more about error handling in a later chapter, but the idea here is that a `Parser` type already explicitly encodes the possibility of failure. It's better for a value of type `Parser a` to have only one vector for errors, and that vector is the parser's ability to encode failure. There may be an edge case that doesn't suit this design preference, but it's a *very* good idea to avoid exceptions or bottoms that aren't explicitly called out as a possibility in your types, whenever possible.

We could modify our program to handle a zero in the denominator case and change it into a parse error:

```
virtuousFraction :: Parser Rational
virtuousFraction = do

  numerator <- decimal
  char '/'
```

```
  denominator <- decimal
  case denominator of
    0 -> fail "Denominator cannot be zero"
    _ -> return (numerator % denominator)
```

Here is our first explicit use of `fail`, which by historical accident is part of the `Monad` type class. Realistically, not all `Monads` have a proper implementation of `fail`, so in more recent versions of GHC, it has been moved out into a `MonadFail` class. For now, it suffices to know that it is our means of returning an error for the `Parser` type.

Let's run our test inputs again but with our more cautious parser:

```
testVirtuous :: IO ()
testVirtuous = do

  let virtuousFraction' =
        parseString virtuousFraction mempty

  print $ virtuousFraction' badFraction
  print $ virtuousFraction' alsoBad

  print $ virtuousFraction' shouldWork
  print $ virtuousFraction' shouldAlsoWork
```

This time, we get a slightly different result at the end:

```
Prelude> testVirtuous
Failure (interactive):1:4: error: Denominator
    cannot be zero, expected: digit
1/0<EOF>
   ^
Failure (interactive):1:3: error: unexpected
    EOF, expected: "/", digit
10<EOF>
  ^
Success (1 % 2)
Success (2 % 1)
```

Now, we have no bottom causing the program to halt, and we get a `Failure` value that explains the cause for the failure. Much better!

Exercise: Unit of success

This should not be unfamiliar at this point, even if you do not understand all the details:

```
Prelude> parseString integer mempty "123abc"
Success 123
Prelude> parseString (integer >> eof) mempty "123abc"
Failure (interactive):1:4: error: expected: digit,
    end of input
123abc<EOF>
   ^
Prelude> parseString (integer >> eof) mempty "123"
Success ()
```

You may have already deduced why it returns `()` as a `Success` result here. It consumes all the input, but there is no result to return from having done so. The result `Success ()` tells you that the parse is successful and consumes the entire input, so there's nothing to return.

What we want you to try now is rewriting the final example so it returns the integer that it parses instead of `Success ()`. It should return the integer successfully when it receives an input with an integer, followed by an EOF, and fail in all other cases:

```
Prelude> parseString (yourFuncHere) mempty "123"
Success 123
Prelude> parseString (yourFuncHere) mempty "123abc"
Failure (interactive):1:4: error: expected: digit,
    end of input
123abc<EOF>
   ^
```

24.5 Haskell's parsing ecosystem

Haskell has several excellent parsing libraries available. `parsec` and `attoparsec` are, perhaps, the two most well known parser combinator libraries in Haskell, but there is also `megaparsec` and others. `aeson` and `cassava` are among the libraries designed for parsing specific types of data (JSON data and CSV data, respectively).

For this chapter, we have opted to use `trifecta`. One reason for that decision is that `trifecta` has error messages that are very easy to read and interpret, unlike some other libraries. Also, `trifecta` does not seem likely to undergo major changes in its fundamental design. Its design is somewhat unusual and complex, but most of the things that make it unusual will be irrelevant to you in this chapter. If you intend to do a lot of parsing in production, you may need to get comfortable using `attoparsec`, as it is particularly known for very speedy parsing; you will see some `attoparsec` (and `aeson`) later in the chapter.

The design of `trifecta` has evolved such that the API[3] is split across two libraries, `parsers`[4] and `trifecta`. The reason for this is that the `trifecta` package itself provides the concrete implementation of the `trifecta` parser as well as `trifecta`-specific functionality, but the `parsers` API is a collection of type classes that abstract over different kinds of things parsers can do. The `Text.Trifecta` module handles exporting what you need to get started from each package, so this information is mostly so you know where to look if you need to start spelunking.

Type classes of parsers

As we note above, `trifecta` relies on the `parsers` library for certain type classes. These type classes abstract over common kinds of things parsers do. We're only going to note a few things here that we'll be seeing in the chapter so that you have a sense of their provenance.

Note that the following is a discussion of code provided for you by the `parsers` library—*you do not need to type this in*!

1. The type class `Parsing` has `Alternative` as a superclass. We'll talk more about `Alternative` in a bit. The `Parsing` type class provides for the functionality needed to describe parsers independently of their input type. A minimal complete instance of this type

[3] API stands for application programming interface. When we write software that relies on libraries or makes requests to a service such as Twitter—basically, software that relies on other software—we rely on a set of defined functions. The API is that set of functions that we use to interface with that software without having to write those functions or worry about their source code. When you look at a library on Hackage, unless you click to view the source code, you're looking at the API for that library.

[4] `http://hackage.haskell.org/package/parsers`

class defines the following functions: `try`, `<?>`, and `notFollowedBy`. Let's start with `try`:

```
-- Text.Parser.Combinators
class Alternative m => Parsing m where
  try :: m a -> m a
```

This takes a parser that may consume input and, on failure, goes back to where it started and fails if we don't consume any input.

It also gives us the function `notFollowedBy`, which does not consume input but allows us to match on keywords by matching on a string of characters that is *not followed by* some thing we do not want to match:

```
notFollowedBy :: Show a => m a -> m ()
-- > noAlpha = notFollowedBy alphaNum
-- > keywordLet =
--     try $ string "let" <* noAlpha
```

2. The `Parsing` type class also includes `unexpected`, which is used to emit an error on an unexpected token, as we saw earlier, and `eof`. The `eof` function only succeeds at the end of input:

```
eof :: m ()
-- > eof =
--         notFollowedBy anyChar
--     <?> "end of input"
```

We'll be seeing more of this one in upcoming sections.

3. The library also defines the type class `CharParsing`, which has `Parsing` as a superclass. This type class handles the parsing of individual characters.

```
-- Text.Parser.Char
class Parsing m => CharParsing m where
```

We've already seen `char` from this class, but it also includes these:

```
-- Parses any single character other
-- than the one provided. Returns
-- the character parsed.
notChar :: Char -> m Char

-- Parser succeeds for any character.
-- Returns the character parsed.
anyChar :: m Char

-- Parses a sequence of characters, returns
-- the string parsed.
string :: String -> m String

-- Parses a sequence of characters
-- represented by a Text value,
-- returns the parsed Text fragment.
text :: Text -> m Text
```

The `parsers` library has much more than this, but for our immediate purposes, these will suffice. The important point is that it defines for us some type classes and basic combinators for common parsing tasks. We encourage you to explore the documentation more on your own.

24.6 Alternative

Let's say we have one parser for numbers and one for alphanumeric strings:

```
Prelude> import Text.Trifecta
Prelude> parseString (some letter) mempty "blah"
Success "blah"
Prelude> parseString integer mempty "123"
Success 123
```

What if we also have a type that could be an `Integer` or a `String`?

```
module AltParsing where

import Control.Applicative
import Text.Trifecta
```

```
type NumberOrString =
  Either Integer String

a = "blah"
b = "123"
c = "123blah789"

parseNos :: Parser NumberOrString
parseNos =
      (Left <$> integer)
  <|> (Right <$> some letter)

main = do
  let p f i =
        parseString f mempty i

  print $ p (some letter) a
  print $ p integer b

  print $ p parseNos a
  print $ p parseNos b

  print $ p (many parseNos) c
  print $ p (some parseNos) c
```

We can read `<|>` as being an *or*, or disjunction, of our two parsers; `many` is *zero or more*, and `some` is *one or more*:

```
Prelude> parseString (some integer) mempty "123"
Success [123]
Prelude> parseString (many integer) mempty "123"
Success [123]
Prelude> parseString (many integer) mempty ""
Success []
Prelude> parseString (some integer) mempty ""
Failure (interactive):1:1: error: unexpected
    EOF, expected: integer
<EOF>
^
```

What we're taking advantage of here with `some`, `many`, and `<|>` is the `Alternative` type class:

```
class Applicative f => Alternative f where
    -- | The identity of '<|>'
    empty :: f a

    -- | An associative binary operation
    (<|>) :: f a -> f a -> f a

    -- | One or more.
    some :: f a -> f [a]
    some v = some_v

      where
        many_v = some_v <|> pure []
        some_v = (fmap (:) v) <*> many_v

    -- | Zero or more.
    many :: f a -> f [a]
    many v = many_v

      where
        many_v = some_v <|> pure []
        some_v = (fmap (:) v) <*> many_v
```

If you use the `:info` command in the REPL after importing `Text.Trifecta` or loading the above module, you'll find that `some` and `many` are defined in `GHC.Base`, because they come from this type class rather than being specific to a particular parser or to the `parsers` library, or even to this particular problem domain.

What if we want to require that each value be separated by a newline? `QuasiQuotes` lets us have a multiline string without the newline separators and use it as a single argument:

```
{-# LANGUAGE QuasiQuotes #-}

module AltParsing where
```

```
import Control.Applicative
import Text.RawString.QQ
import Text.Trifecta

type NumberOrString =
  Either Integer String

eitherOr :: String
eitherOr = [r|
123
abc
456
def
|]
```

QuasiQuotes

Above, the `[r|` begins a quasiquoted[5] section, using the quasiquoter named `r`. Note that we have to enable the `QuasiQuotes` language extension to use this syntax. At the time of writing, `r` is defined in `raw-strings-qq` version 1.1 as follows:

```
r :: QuasiQuoter
r = QuasiQuoter {
    -- Extracted from dead-simple-json.
    quoteExp  =
      return . LitE . StringL
             . normaliseNewlines,

    -- error messages elided
    quotePat  =
      \_ -> fail "some error message"

    quoteType =
      \_ -> fail "some error message"
    quoteDec  =
      \_ -> fail "some error message"
```

[5] There's a rather nice wiki page and tutorial example at: `https://wiki.haskell.org/Quasiquotation`.

This is a macro that lets us write arbitrary text inside of the block that begins with `[r|` and ends with `|]`. This specific quasiquoter allows us to write multiline strings without having to manually escape the newlines. The quasiquoter generates the following string for us:

```
"\n\
\123\n\
\abc\n\
\456\n\
\def\n"
```

Not as nice right? As it happens, if you want to see what a quasiquoter or Template Haskell[6] is generating at compile-time, you can enable the `-ddump-splices` flag to see what it does. Here's an example using a minimal stub file:

```
{-# LANGUAGE QuasiQuotes #-}

module Quasimodo where

import Text.RawString.QQ

eitherOr :: String
eitherOr = [r|
123
abc
456
def
|]
```

Then, in GHCi, we use the `:set` command to turn on the splice dumping flag, so we can see what the quasiquoter generates:

```
Prelude> :set -ddump-splices
Prelude> :l code/quasi.hs
[1 of 1] Compiling Quasimodo
code/quasi.hs:(8,12)-(12,2): Splicing expression
    "\n\
```

[6] https://wiki.haskell.org/Template_Haskell

```
  \123\n\
  \abc\n\
  \456\n"
======>
  "\n\
  \123\n\
  \abc\n\
  \456\n"
```

Right, so back to the parser we were going to write!

Return to Alternative

All right, we return now to our `AltParsing` module. We're going to use this fantastic function:

```
parseNos :: Parser NumberOrString
parseNos =
      (Left <$> integer)
  <|> (Right <$> some letter)
```

And rewrite `main` to apply that to the `eitherOr` value:

```
main = do
  let p f i = parseString f mempty i
  print $ p parseNos eitherOr
```

Note that we lift `Left` and `Right` over their arguments. This is because there is `Parser` structure between the (potential) value obtained by running the parser and what the data constructor expects. A value of type `Parser Char` is a parser that will *possibly* produce a `Char` value if it is given an input that doesn't cause it to fail. The type of `some letter` is the following:

```
Prelude> import Text.Trifecta
Prelude> :t some letter
some letter :: CharParsing f => f [Char]
```

However, for our purposes, we can say that the type is specifically the `Parser` type from `trifecta`:

```
Prelude> someLetter = some letter :: Parser [Char]
Prelude> someLetter = some letter :: Parser String
```

If we try to mash a data constructor expecting a `String` and our parser-of-string together like a kid playing with action figures, we get a type error:

```
data MyName = MyName String deriving Show
```

```
Prelude> MyName someLetter


• Couldn't match type ‘Parser String’ with ‘[Char]’
  Expected type: String
    Actual type: Parser String
• In the first argument of ‘MyName’,
    namely ‘someLetter’
  In the expression: MyName someLetter
```

Unless we lift it over the `Parser` structure, since `Parser` is a `Functor`!

```
Prelude> :info Parser
...content elided...
instance Monad Parser
instance Functor Parser
instance Applicative Parser
instance Monoid a => Monoid (Parser a)

instance Errable Parser
instance DeltaParsing Parser
instance TokenParsing Parser
instance Parsing Parser
instance CharParsing Parser
```

We should need an `fmap`, right?

```
Prelude> :t MyName <$> someLetter
MyName <$> someLetter :: Parser MyName
Prelude> :t MyName `fmap` someLetter
MyName `fmap` someLetter :: Parser MyName
```

Then, running either of them:

```
Prelude> parseString someLetter mempty "Chris"
Success "Chris"
Prelude> myNameParser = MyName <$> someLetter
Prelude> parseString myNameParser mempty "Chris"
Success (MyName "Chris")
```

Cool.

Back to our original code, which spits out an error when we run it:

```
Prelude> main
Failure (interactive):1:1: error: expected:
  integer, letter
```

It's easier to see why if we look at the test string:

```
Prelude> eitherOr
"\n123\nabc\n456\ndef\n"
```

One way to fix this is to amend the quasiquoted string:

```
eitherOr :: String
eitherOr = [r|123
abc
456
def
|]
```

What if we want to permit a newline before attempting to parse strings or integers?

```
eitherOr :: String
eitherOr = [r|
123
abc
456
def
|]
```

```
parseNos :: Parser NumberOrString
parseNos =
  skipMany (oneOf "\n")
  >>
      (Left <$> integer)
  <|> (Right <$> some letter)

main = do
  let p f i = parseString f mempty i
  print $ p parseNos eitherOr
```

```
Prelude> main
Success (Left 123)
```

OK, but we'd like to keep parsing after each line. If we try the obvious thing and use `some` to ask for one-or-more results, we'll get a somewhat mysterious error:

```
Prelude> parseString (some parseNos) mempty eitherOr
Failure (interactive):6:1: error: unexpected
    EOF, expected: integer, letter
<EOF>
^
```

The issue here is that while `skipMany` lets us skip zero or more times, it means we start the next run of the parser before we hit EOF. This means it *expects* us to match an integer or some letters after having seen the newline character after `"def"`. We can simply amend the input:

```
eitherOr :: String
eitherOr = [r|
123
abc
456
def|]
```

Our previous attempt will now work fine:

```
Prelude> parseString (some parseNos) mempty eitherOr
Success [Left 123,Right "abc",Left 456,Right "def"]
```

If we're dissatisfied with simply changing the rules of the game, there are a couple ways we can make our parser cope with spurious terminal newlines. One is to add another `skipMany` rule after we parse our value:

```
parseNos :: Parser NumberOrString
parseNos = do
  skipMany (oneOf "\n")

  v <-     (Left <$> integer)
       <|> (Right <$> some letter)

  skipMany (oneOf "\n")
  return v
```

Another option is to keep the previous version of the parser that skips a potential leading newline:

```
parseNos :: Parser NumberOrString
parseNos =
  skipMany (oneOf "\n")
  >>
      (Left <$> integer)
  <|> (Right <$> some letter)
```

But then tokenize it with the default `token` behavior:

```
Prelude> parseString (some (token parseNos)) mempty eitherOr
Success [Left 123,Right "abc",Left 456,Right "def"]
```

We'll explain soon what this token stuff is about, but we want to be a bit careful here, as token parsers and character parsers are different sorts of things. What applying `token` to `parseNos` did for us here is make it *optionally* consume trailing whitespace we don't care about, where whitespace includes newline characters.

Exercise: Try try

Make a parser, using the existing fraction parser plus a new decimal parser, that can parse either decimals or fractions. You'll want to use

`<|>` from `Alternative` to combine the… alternative parsers. If you find this too difficult, write a parser that parses straightforward integers *or* fractions. Make a datatype that contains either an integer or a rational and use that datatype as the result of the parser. Or use `Either`. Run free, grasshopper.

Hint: we've not explained it yet, but you may want to try `try`.

24.7 Parsing configuration files

For our next few examples, we'll be using the INI[7] configuration file format, partly because it's an informal standard, so we can play fast and loose for learning and experimentation purposes. We're also using INI, because it's relatively uncomplicated.

Here's a teensy example of an INI config file:

```
; comment
[section]
host=wikipedia.org
alias=claw
```

The above contains a comment, which contributes nothing to the data parsed out of the configuration file, but which may provide context to the settings being configured. It's followed by a section header named `section` that contains two settings: one named `host`, with the value `wikipedia.org`, and another named `alias`, with the value `claw`.

We'll begin this example with our pragmas, module declaration, and imports:

```
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE QuasiQuotes       #-}

module Data.Ini where

import Control.Applicative
import Data.ByteString (ByteString)
import Data.Char (isAlpha)
import Data.Map (Map)
```

[7] INI is an informal standard for configuration files on some platforms. The name comes from the file extension, `.ini`, short for "initialization."

```
import qualified Data.Map as M
import Data.Text (Text)
import qualified Data.Text.IO as TIO
import Test.Hspec

import Text.RawString.QQ
-- parsers 0.12.3, trifecta 1.5.2
import Text.Trifecta
```

`OverloadedStrings` and `QuasiQuotes` should be familiar by now.

When writing parsers in Haskell, it's often easiest to work in terms of smaller parsers that deal with a sub-problem of the overall parsing problem you're solving, then combine them into the final parser. This isn't a perfect recipe for understanding your parser, but being able to compose them straightforwardly like functions is pretty nifty. Let's start by creating a test input for an INI header, a datatype, and then the parser for it:

```
headerEx :: ByteString
headerEx = "[blah]"

-- "[blah]" -> Section "blah"
newtype Header =
  Header String
  deriving (Eq, Ord, Show)

parseBracketPair :: Parser a -> Parser a
parseBracketPair p =
  char '[' *> p <* char ']'
```

These operators mean the brackets will be parsed and then discarded, but the `p` will remain as our result:

```
parseHeader :: Parser Header
parseHeader =
  parseBracketPair (Header <$> some letter)
```

Here, we've combined two parsers in order to parse a `Header`. We can experiment with each of them in the REPL. First, we'll examine the types of the `some letter` parser we pass to `parseBracketPair`:

```
Prelude> :t some letter
some letter :: CharParsing f => f [Char]
Prelude> h = Header <$> some letter
Prelude> :t h
Header <$> some letter :: CharParsing f => f Header
Prelude> slp = h :: Parser Header
```

The first type is some parser that can understand characters that will produce a `String` value if it succeeds. The second type is the same, but it produces a `Header` value instead of a `String`. Parser types in Haskell almost always encode the possibility of failure—we'll cover how later in this chapter. The third type gives us a concrete `Parser` type from `trifecta` in place of the polymorphic type `f`.

The `letter` function parses a single character, while `some letter` parses one or more characters. We need to wrap the `Header` constructor around that so that our result there—whatever letters might be inside the brackets, the `p` of `parseBracketPair`—will be labeled as the `Header` of the file in the final parse.

Next, `assignmentEx` is just some test input, so we can begin kicking around our parser. The type synonyms are to make the types more readable, as well. Nothing too special here:

```
assignmentEx :: ByteString
assignmentEx = "woot=1"

type Name = String
type Value = String
type Assignments = Map Name Value

parseAssignment :: Parser (Name, Value)
parseAssignment = do
  name <- some letter
  _ <- char '='
  val <- some (noneOf "\n")
  skipEOL -- important!
  return (name, val)
```

```
-- | Skip end of line and
--   whitespace beyond.
skipEOL :: Parser ()
skipEOL = skipMany (oneOf "\n")
```

Let us explain `parseAssignment` step by step. For parsing the initial key or name of an assignment, we parse one or more letters:

```
name <- some letter
```

Then, we parse and throw away the "=" used to separate keys and values:

```
_ <- char '='
```

Then, we parse one or more characters, as long as they aren't a newline. This is so that letters, numbers, and whitespace are permitted:

```
val <- some (noneOf "\n")
```

We skip "end-of-line" until we stop getting newline characters:

```
skipEOL -- important!
```

This is so we can delineate the end of assignments and parse more than one assignment in a straightforward manner. Consider an alternative variant of this same parser that doesn't have `skipEOL`:

```
parseAssignment' :: Parser (Name, Value)
parseAssignment' = do
  name <- some letter
  _ <- char '='
  val <- some (noneOf "\n")
  return (name, val)
```

Then, trying out this variant of the parser:

```
Prelude> spa' = some parseAssignment'
Prelude> s = "key=value\nblah=123"
Prelude> parseString spa' mempty s
Success [("key","value")]
```

Pity. Can't parse the second assignment. But the first version that includes the `skipEOL` should work:

```
Prelude> spa = some parseAssignment
Prelude> parseString spa mempty s
Success [("key","value"),("blah","123")]
Prelude> d = "key=value\n\n\ntest=data"
Prelude> parseString spa mempty d
Success [("key","value"),("test","data")]
```

We have to skip the one-or-more newline characters separating the first and second assignments in order for the rerun of the assignment parser to begin successfully parsing the letters that make up the key of the second assignment. Happy-making, right?

We finish things off for `parseAssignment` by tupling the name and value together and re-embedding the result in the `Parser` type:

```
return (name, val)
```

Then, we deal with INI comments—that is, we skip them in the parser and discard the data:

```
commentEx :: ByteString
commentEx =
  "; last modified 1 April\
  \ 2001 by John Doe"

commentEx' :: ByteString
commentEx' =
  "; blah\n; woot\n  \n;hah"

-- | Skip comments starting at the
--   beginning of the line.
skipComments :: Parser ()
skipComments =
  skipMany (do _ <- char ';' <|> char '#'
               skipMany (noneOf "\n")
               skipEOL)
```

We made a couple of comment examples for testing the parser. Note that comments can begin with `#` or `;`.

Next, we need section parsing. We'll make some data for testing that out, as we did with comments above. This is also where we'll put that `QuasiQuotes` extension to use, allowing us to make multiline strings nicer to write:

```
sectionEx :: ByteString
sectionEx =
  "; ignore me\n[states]\nChris=Texas"

sectionEx' :: ByteString
sectionEx' = [r|
; ignore me
[states]
Chris=Texas
|]

sectionEx'' :: ByteString
sectionEx'' = [r|
; comment
[section]
host=wikipedia.org
alias=claw

[whatisit]
red=intoothandclaw
|]
```

Then, we get into the section parsing itself:

```
data Section =
  Section Header Assignments
  deriving (Eq, Show)

newtype Config =
  Config (Map Header Assignments)
  deriving (Eq, Show)
```

```
skipWhitespace :: Parser ()
skipWhitespace =
  skipMany (char ' ' <|> char '\n')

parseSection :: Parser Section
parseSection = do
  skipWhitespace
  skipComments

  h <- parseHeader
  skipEOL

  assignments <- some parseAssignment
  return $
    Section h (M.fromList assignments)
```

Above, we define datatypes for a section and an entire INI config. You'll notice that `parseSection` skips both whitespace and comments now. And it returns the parsed section with the header (that's the `h`) and a map of assignments:

```
*Data.Ini> parseByteString parseSection mempty sectionEx
Success (Section (Header "states")
                 (fromList [("Chris","Texas")]))
```

So far, so good. Next, let's roll the sections up into a `Map` that keys section data by section name, with the values being more `Maps` of assignment names mapped to their values. We use `foldr` to aggregate the list of sections into a single `Map` value:

```
rollup :: Section
       -> Map Header Assignments
       -> Map Header Assignments
rollup (Section h a) m =
  M.insert h a m
```

```
parseIni :: Parser Config
parseIni = do
  sections <- some parseSection
  let mapOfSections =
        foldr rollup M.empty sections
  return (Config mapOfSections)
```

After you load this code into your REPL, try running this:

```
parseByteString parseIni mempty sectionEx
```

And then compare it to the output of this, which you saw above:

```
parseByteString parseSection mempty sectionEx
```

Now, we'll put these things together. We're interested in whether our parsers do what they should do rather than parsing an actual INI file, so we'll have `main` run some `hspec` tests. We'll use a helper function, `maybeSuccess`, as part of the tests:

```
maybeSuccess :: Result a -> Maybe a
maybeSuccess (Success a) = Just a
maybeSuccess _ = Nothing

main :: IO ()
main = hspec $ do

  describe "Assignment Parsing" $
    it "can parse a simple assignment" $ do
      let m = parseByteString
              parseAssignment
              mempty assignmentEx
          r' = maybeSuccess m

      print m
      r' `shouldBe` Just ("woot", "1")
```

```
describe "Header Parsing" $
  it "can parse a simple header" $ do
    let m =
          parseByteString parseHeader
          mempty headerEx
        r' = maybeSuccess m

    print m
    r' `shouldBe` Just (Header "blah")

describe "Comment parsing" $
  it "Skips comment before header" $ do
    let p = skipComments >> parseHeader
        i = "; woot\n[blah]"
        m = parseByteString p mempty i
        r' = maybeSuccess m

    print m
    r' `shouldBe` Just (Header "blah")

describe "Section parsing" $
  it "can parse a simple section" $ do
    let m = parseByteString parseSection
            mempty sectionEx
        r' = maybeSuccess m

        states =
          M.fromList [("Chris", "Texas")]

        expected' =
          Just (Section (Header "states")
                        states)
    print m
    r' `shouldBe` expected'

describe "INI parsing" $
  it "Can parse multiple sections" $ do
    let m =
          parseByteString parseIni
          mempty sectionEx''
```

```
    r' = maybeSuccess m
    sectionValues =
      M.fromList
      [ ("alias", "claw")
      , ("host", "wikipedia.org")]

    whatisitValues =
      M.fromList
      [("red", "intoothandclaw")]

    expected' =
      Just (Config
            (M.fromList
             [ (Header "section"
             , sectionValues)
             , (Header "whatisit"
             , whatisitValues)]))

print m
r' `shouldBe` expected'
```

We leave it to you to run this and experiment with it.

24.8 Character and token parsers

All right, that was a lot of code. Let's all step back and take a deep breath.

You probably have some idea by now of what we mean by tokenizing, but the time has come for more detail. Tokenization is a handy parsing tactic, so it's baked into some of the library functions we've been using. It's worth diving in and exploring what it means.

Traditionally, parsing has been done in two stages, lexing and parsing. Characters from a stream will be fed into the lexer, which will then emit tokens on demand to the parser until it has no more to emit.[8] The parser then structures the stream of tokens into a tree, commonly called an "abstract syntax tree" or AST:

[8]Lexers and tokenizers are similar, separating a stream of text into tokens based on indicators such as whitespace or newlines; lexers often attach some context to the tokens, whereas tokenizers typically do not.

```
-- hand-wavy types: Stream because
-- production-grade parsers in Haskell
-- won't use [] for performance reasons
lexer :: Stream Char -> Stream Token
parser :: Stream Token -> AST
```

Lexers are simpler, typically performing parses that don't require looking ahead into the input stream by more than one character or token at a time. Lexers are at times called tokenizers. Lexing is sometimes done with *regular expressions*, but a parsing library in Haskell will usually intend that you do your lexing and parsing with the same API. Lexers (or tokenizers) and parsers have a lot in common, being primarily differentiated by their purposes and classes of grammar.

Insert tokens to play

Let's play around with some things to see what tokenizing does for us:

```
Prelude> parseString (some digit) mempty "123 456"
Success "123"
Prelude> parseString (some (some digit)) mempty "123 456"
Success ["123"]

Prelude> parseString (some integer) mempty "123"
Success [123]
Prelude> parseString (some integer) mempty "123456"
Success [123456]
```

The problem here is that if we want to recognize 123 and 456 as independent strings, we need some kind of separator. We can go ahead and do that manually, but the tokenizers in `parsers` can do it for you, too, also handling a mixture of whitespace and newlines:

```
Prelude> parseString (some integer) mempty "123 456"
Success [123,456]
Prelude> parseString (some integer) mempty "123\n\n  456"
Success [123,456]
```

Or even spaces and newlines interleaved:

```
Prelude> parseString (some integer) mempty "123 \n \n 456"
Success [123,456]
```

But simply applying `token` to `digit` doesn't do what you think:

```
Prelude> s = "123 \n \n 456"
Prelude> t = token
Prelude> ps = parseString
Prelude> ps (t (some digit)) mempty s
Success "123"
Prelude> ps (t (some (t digit))) mempty s
Success "123456"

Prelude> ps (some decimal) mempty s
Success [123]
Prelude> ps (some (t decimal)) mempty s
Success [123,456]
```

Compare that to the `integer` function, which is already a tokenizer:

```
Prelude> parseString (some integer) mempty "1\n2\n 3\n"
Success [1,2,3]
```

We can write a tokenizing parser such as `some integer` like this:

```
p' :: Parser [Integer]
p' = some $ do
  i <- token (some digit)
  return (read i)
```

And we can compare the output of that to the output of applying `token` to `digit`:

```
Prelude> s = "1\n2\n3"
Prelude> parseString p' mempty s
Success [1,2,3]
Prelude> parseString (token (some digit)) mempty s
Success "1"
Prelude> parseString (some (token (some digit))) mempty s
Success ["1","2","3"]
```

You'll want to think carefully about the scope in which you're tokenizing, as well:

```
Prelude> tknWhole = token $ char 'a' >> char 'b'
Prelude> parseString tknWhole mempty "a b"
Failure (interactive):1:2: error: expected: "b"
a b<EOF>
 ^
Prelude> parseString tknWhole mempty "ab ab"
Success 'b'
Prelude> parseString (some tknWhole) mempty "ab ab"
Success "bb"
```

If we want that first example to work, we need to tokenize the parse of the first character, not the whole a-then-b parse:

```
Prelude> tknCharA = (token (char 'a')) >> char 'b'
Prelude> parseString tknCharA mempty "a b"
Success 'b'
Prelude> parseString (some tknCharA) mempty "a ba b"
Success "bb"
Prelude> parseString (some tknCharA) mempty "a b a b"
Success "b"
```

The last example stops at the first `"a b"` parse, because the parser doesn't say anything about a space after `'b'`, and the tokenization behavior only applies to what follows `'a'`. We can tokenize both character parsers, though:

```
Prelude> tokenA = token (char 'a')
Prelude> tokenB = token (char 'b')
Prelude> tknBoth = tokenA >> tokenB
Prelude> parseString (some tknBoth) mempty "a b a b"
Success "bb"
```

A mild warning: don't get too tokenization happy. Try to make it coarse-grained and selective. Overuse of tokenizing parsers or mixture with character parsers can make your parser slow or hard to understand. Use your judgment. Keep in mind that tokenization isn't exclusively about whitespace; it's about ignoring noise, so you can focus on the structures you are parsing.

24.9 Polymorphic parsers

If we take the time to assert polymorphic types for our parsers, we can get parsers that can be run using `attoparsec`, `trifecta`, `parsec`, or anything else that has implemented the necessary type classes. Let's give it a whirl, shall we?

```
{-# LANGUAGE OverloadedStrings #-}

module Text.Fractions where

import Control.Applicative
import Data.Attoparsec.Text (parseOnly)
import Data.Ratio ((%))
import Data.String (IsString)
import Text.Trifecta

badFraction :: IsString s => s
badFraction = "1/0"

alsoBad :: IsString s => s
alsoBad = "10"

shouldWork :: IsString s => s
shouldWork = "1/2"

shouldAlsoWork :: IsString s => s
shouldAlsoWork = "2/1"

parseFraction :: (Monad m, TokenParsing m)
              => m Rational
parseFraction = do
  numerator <- decimal
  _ <- char '/'

  denominator <- decimal
  case denominator of
    0 -> fail "Denominator cannot be zero"
    _ -> return (numerator % denominator)
```

We've left some type class-constrained polymorphism in our type signatures for flexibility. Our `main` will run both `attoparsec` and `trifecta` versions for us, so we can compare the outputs directly:

```
main :: IO ()
main = do
  -- parseOnly is Attoparsec
  let attoP = parseOnly parseFraction

  print $ attoP badFraction
  print $ attoP shouldWork
  print $ attoP shouldAlsoWork
  print $ attoP alsoBad

  -- parseString is Trifecta
  let p f i =
        parseString f mempty i

  print $ p parseFraction badFraction
  print $ p parseFraction shouldWork
  print $ p parseFraction shouldAlsoWork
  print $ p parseFraction alsoBad
```

```
Prelude> main
Left "Failed reading: Denominator cannot be zero"
Right (1 % 2)
Right (2 % 1)
Left "\"/\": not enough input"
Failure (interactive):1:4: error: Denominator
    cannot be zero, expected: digit
1/0<EOF>
   ^
Success (1 % 2)
Success (2 % 1)
Failure (interactive):1:3: error: unexpected
    EOF, expected: "/", digit
10<EOF>
  ^
```

See what we meant earlier about the error messages?

It's not perfect and could bite you

While the polymorphic parser combinators in the `parsers` library enable you to write parsers that can then be run with various parsing libraries, this doesn't free you from having to understand the particularities of each. In general, `trifecta` tries to match `parsec`'s behaviors in most respects, the latter of which is more extensively documented.

Failure and backtracking

Returning to our cursor model of parsers, backtracking is returning the cursor to where it was before a failing parser consumed input. In some cases, it can be a little confusing to debug the same error in two different runs of the same parser doing essentially the same things in `trifecta`, `parsec`, and `attoparsec`, because the errors themselves might be different. Let's consider an example of this. We use `OverloadedStrings`, so that we can use string literals as if they were `ByteStrings`, when testing `attoparsec`:

```
{-# LANGUAGE OverloadedStrings #-}

module BT where

import Control.Applicative
import qualified Data.Attoparsec.ByteString
  as A
import Data.Attoparsec.ByteString
  (parseOnly)

import Data.ByteString (ByteString)
import Text.Trifecta hiding (parseTest)
import Text.Parsec (Parsec, parseTest)


trifP :: Show a
      => Parser a
      -> String -> IO ()

trifP p i =
  print $ parseString p mempty i
```

Helper function to run a `trifecta` parser and print the result:

```
parsecP :: (Show a)
        => Parsec String () a
        -> String -> IO ()
parsecP = parseTest
```

Helper function to run a `parsec` parser and print the result:

```
attoP :: Show a
      => A.Parser a
      -> ByteString -> IO ()
attoP p i =
  print $ parseOnly p i
```

Helper function for `attoparsec`—same deal as before:

```
nobackParse :: (Monad f, CharParsing f)
            => f Char
nobackParse =
       (char '1' >> char '2')
   <|> char '3'
```

Here's our first parser. It attempts to parse `'1'` followed by `'2'` *or* `'3'`. This parser does *not* backtrack:

```
tryParse :: (Monad f, CharParsing f)
         => f Char
tryParse =
       try (char '1' >> char '2')
   <|> char '3'
```

This parser has similar behavior to the previous one, except it *backtracks* if the first parse fails. *Backtracking* means that the input cursor returns to where it was before the failed parser consumed input:

```
main :: IO ()
main = do
  -- trifecta
  trifP nobackParse "13"
  trifP tryParse "13"
```

```
  -- parsec
  parsecP nobackParse "13"
  parsecP tryParse "13"

  -- attoparsec
  attoP nobackParse "13"
  attoP tryParse "13"
```

The error messages you get from each parser are going to vary a bit. This isn't because they're wildly different but is mostly due to how they attribute errors. You should see something like this:

```
Prelude> main
Failure (interactive):1:2:
  error: expected: "2"
13<EOF>
 ^
Failure (interactive):1:1: error:
  expected: "3"
13<EOF>
^
parse error at (line 1, column 2):
unexpected "3"
expecting "2"
parse error at (line 1, column 2):
unexpected "3"
expecting "2"
Left "\"3\": satisfyElem"
Left "\"3\": satisfyElem"
```

Conversely, if you try the valid inputs `"12"` and `"3"` with `nobackParse` and each of the three parsers, you should see all of them succeed.

This can be confusing. When you add backtracking to a parser, error attribution can become more complicated, at times. To avoid this, consider using the `<?>` operator to annotate parse rules any time you use `try`:[9]

[9] See Edward Z. Yang, *Parsec "try a <|> b" considered harmful*: `http://blog.ezyang.com/2014/05/parsec-try-a-or-b-considered-harmful/`.

```
tryAnnot :: (Monad f, CharParsing f)
         => f Char
tryAnnot =
       (try (char '1' >> char '2')
       <?> "Tried 12")
   <|> (char '3' <?> "Tried 3")
```

Then, run this in the REPL:

```
Prelude> trifP tryAnnot "13"
Failure (interactive):1:1: error: expected:
  Tried 12, Tried 3
13<EOF>
^
```

Now, the error will list the parses it attempts before it fails. In your own parsers, you'll want to make the annotations more informative than what we demonstrate.

24.10 Marshalling from an AST to a datatype

Fair warning: this section relies on a little more background knowledge from you than previous sections have. If you are not a person who already has some programming experience, the following may not seem terribly useful to you, and there may be some unfamiliar terminology and concepts.

The act of parsing, in a sense, is a means of necking down the cardinality of our inputs to the set of things our programs have a sensible answer for. It's unlikely you can do something meaningful and domain-specific when your input type is `String`, `Text`, or `ByteString`. However, if you can parse one of those types into something structured, rejecting bad inputs along the way, then you might be able to write a proper program. One of the mistakes programmers make in writing programs that handle text is in allowing their data to *stay* in the textual format, doing mind-bending backflips to cope with the unstructured nature of textual inputs.

In some cases, the act of parsing isn't enough. You might have a sort of AST or structured representation of what gets parsed, but from there, you might also expect that AST or representation to take

a particular *form*. This means we want to narrow the cardinality and be *even more specific* about how our data looks. Often, this second step is called *unmarshalling* our data. Similarly, *marshalling* is the act of preparing data for serialization, whether via memory alone (foreign function interface boundary) or over a network interface.

The whole idea here is that you have two pipelines for your data:

```
    Text -> Structure  -> Meaning
--  parse -> unmarshall

   Meaning  -> Structure -> Text
--  marshall -> serialize
```

There's more than one way to accomplish this, but we'll show you a commonly used library and how it uses this two-stage pipeline in its API.

Marshalling and unmarshalling JSON data

`aeson` is presently the most popular JSON[10] library in Haskell. One of the things that confuses programmers coming to Haskell from Python, Ruby, Clojure, JavaScript, or similar languages, is that there's usually no unmarshall/marshall step when dealing with JSON. Instead, the raw JSON AST will be represented directly as an untyped blob of data. Users of typed languages are more likely to have encountered something like this. We'll be using `aeson` 0.10.0.0 for the following examples:

```
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE QuasiQuotes       #-}

module Marshalling where

import Data.Aeson
import Data.ByteString.Lazy (ByteString)
import Text.RawString.QQ
```

[10]JSON stands for JavaScript Object Notation. JSON is, for better or worse, a very common open-standard data format used to transmit data, especially between browsers and servers. As such, dealing with JSON is a common programming task, so you might as well get used to it now.

```
sectionJson :: ByteString
sectionJson = [r|
{ "section":  {"host": "wikipedia.org"},
  "whatisit": {"red": "intoothandclaw"}
}
|]
```

Note that we're saying the type of `sectionJson` is a *lazy* `ByteString`. If you get strict and lazy `ByteString` types mixed up, you'll get errors. For example, this is what happens when you provide a *strict* `ByteString` when a *lazy* one is expected:

```
<interactive>:10:8:
Couldn't match expected type
  Data.ByteString.Lazy.Internal.ByteString
  with actual type ByteString

NB:
Data.ByteString.Lazy.Internal.ByteString
  is defined in
  Data.ByteString.Lazy.Internal

ByteString
  is defined in
  Data.ByteString.Internal
```

The *actual type* is what we provide. The *expected type* is what the types want. The `NB:` in the type error stands for *nota bene*. Either we used the wrong code (so the expected type needs to change), or we provided the wrong values (the actual type, our types/values, need to change). You can reproduce this error by making a deliberate mistake in the marshalling module.

Change the import of the `ByteString` type constructor from:

```
import Data.ByteString.Lazy (ByteString)
```

Into:

```
import Data.ByteString (ByteString)
```

And this is what happens when you provide a *lazy* `ByteString` when a *strict* one is expected:

```
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE QuasiQuotes       #-}

module WantedStrict where

import Data.Aeson
import Data.ByteString.Lazy (ByteString)
import Text.RawString.QQ

sectionJson :: ByteString
sectionJson = [r|
{ "section":  {"host": "wikipedia.org"},
  "whatisit": {"red": "intoothandclaw"}
}
|]

main = do
  let blah :: Maybe Value
      blah = decodeStrict sectionJson
  print blah
```

You'll get the following type error if you load that up:

```
code/wantedStrictGotLazy.hs:19:27:
Couldn't match expected type
  ‘Data.ByteString.Internal.ByteString’
  with actual type ‘ByteString’
NB:
‘Data.ByteString.Internal.ByteString’
  is defined in ‘Data.ByteString.Internal’
‘ByteString’ is defined in
  ‘Data.ByteString.Lazy.Internal’

In the first argument of ‘decodeStrict’,
  namely ‘sectionJson’
In the expression: decodeStrict sectionJson
```

The more useful information is in the `NB:`, where the internal modules are mentioned. The key is to remember that *actual type* means "your code," *expected type* means "what the compiler expects," and that the `ByteString` module that doesn't have `Lazy` in the name is the strict version. We can modify our code a bit to get nicer type errors:

```
-- replace the (ByteString)
-- import with these
import qualified Data.ByteString as BS
import qualified Data.ByteString.Lazy
  as LBS

-- edit the type sig for this one
sectionJson :: LBS.ByteString
```

Then, we'll get the following type error, instead:

```
Couldn't match expected type ‘BS.ByteString’
               with actual type ‘LBS.ByteString’

NB: ‘BS.ByteString’ is defined in
     ‘Data.ByteString.Internal’

    ‘LBS.ByteString’ is defined in
    ‘Data.ByteString.Lazy.Internal’
In the first argument of ‘decodeStrict’,
  namely ‘sectionJson’
In the expression: decodeStrict sectionJson
```

This is helpful, because we have both versions available as qualified modules. You may not always be so fortunate and will need to remember which is which.

Back to the JSON

Let's get back to handling JSON. The most common functions for using `aeson` are the following:

```
Prelude> import Data.Aeson
```

```
Prelude> :t encode
encode :: ToJSON a => a -> LBS.ByteString
Prelude> :t decode
decode :: FromJSON a => LBS.ByteString -> Maybe a
```

These functions are sort of eliding the intermediate step that passes through the `Value` type in `aeson`, which is a JSON AST datatype—sort of, because you can decode the raw JSON data into a `Value`, anyway:

```
Prelude> decode sectionJson :: Maybe Value
Just (Object (fromList [
("whatisit",
  Object (fromList [("red",
  String "intoothandclaw")])),
("section",
  Object (fromList [("host",
  String "wikipedia.org")]))]))
```

Not, uh, super pretty. We'll figure out something nicer in a moment. Also, do not forget to assert a type, or the type-defaulting in GHCi will do silly things:

```
Prelude> decode sectionJson
Nothing
```

Now, what if we do want a nicer representation for this JSON noise? Well, let's define our datatypes and see if we can decode the JSON into our type:

```
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE QuasiQuotes       #-}

module Marshalling where

import Control.Applicative
import Data.Aeson
import Data.ByteString.Lazy (ByteString)
import qualified Data.Text as T
import Data.Text (Text)
import Text.RawString.QQ
```

```
sectionJson :: ByteString
sectionJson = [r|
{ "section":  {"host": "wikipedia.org"},
  "whatisit": {"red": "intoothandclaw"}
}
|]

data TestData =
  TestData {
    section :: Host
  , what :: Color
  } deriving (Eq, Show)

newtype Host =
  Host String
  deriving (Eq, Show)

type Annotation = String

data Color =
    Red Annotation
  | Blue Annotation
  | Yellow Annotation
  deriving (Eq, Show)

main = do
  let d :: Maybe TestData
      d = decode sectionJson
  print d
```

This will, in fact, net you a type error that complains that there isn't an instance of `FromJSON` for `TestData`. Which is true! GHC has no idea how to unmarshall JSON data (in the form of a `Value`) into a `TestData` value. Let's add an instance, so it knows how:

```
instance FromJSON TestData where
  parseJSON (Object v) =
    TestData <$> v .: "section"
             <*> v .: "whatisit"
  parseJSON _ =
    fail "Expected an object for TestData"

instance FromJSON Host where
  parseJSON (Object v) =
    Host <$> v .: "host"
  parseJSON _ =
    fail "Expected an object for Host"

instance FromJSON Color where
  parseJSON (Object v) =
        (Red <$> v .: "red")
    <|> (Blue <$> v .: "blue")
    <|> (Yellow <$> v .: "yellow")
  parseJSON _ =
    fail "Expected an object for Color"
```

Also, note that you can use quasiquotes to avoid having to escape quotation marks in the REPL, as well:

```
Prelude> :set -XOverloadedStrings
Prelude> decode "{\"blue\": \"123\"}" :: Maybe Color
Just (Blue "123")
Prelude> :set -XQuasiQuotes
Prelude> decode [r|{"red": "123"}|] :: Maybe Color
Just (Red "123")
```

To relate what we just did back to the relationship between parsing and marshalling, the idea is that our `FromJSON` instance accepts the `Value` type, and `ToJSON` instances generate the `Value` type, closing the following loop:

```
-- FromJSON
ByteString -> Value -> yourType
-- parse -> unmarshall

-- ToJSON
  yourType  -> Value -> ByteString
-- marshall -> serialize
```

The definition of `Value` at the time of writing is the following:

```
-- | A JSON value represented
--   as a Haskell value.
data Value = Object !Object
           | Array !Array
           | String !Text
           | Number !Scientific
           | Bool !Bool
           | Null
           deriving (Eq, Read, Show,
                     Typeable, Data)
```

What if we want to unmarshall something that could be a `Number` or a `String`?

```
data NumberOrString =
    Numba Integer
  | Stringy Text
  deriving (Eq, Show)

instance FromJSON NumberOrString where
  parseJSON (Number i) = return $ Numba i
  parseJSON (String s) = return $ Stringy s
  parseJSON _ =
    fail "NumberOrString must\
         \ be number or string"
```

This won't quite work, at first. The trouble is that JSON (and JavaScript, as it happens) only has one numeric type, and that type is an IEEE-754 float. JSON (and JavaScript, terrifyingly) have no

integral types or integers, so `aeson` has to pick *one* representation that works for all possible JSON numbers. The most precise way to do that is the `Scientific` type, which is an arbitrarily precise numerical type (you may remember this from way back in Chapter 4, Basic datatypes). So, we need to convert from a `Scientific` to an `Integer`:

```
import Control.Applicative
import Data.Aeson
import Data.ByteString.Lazy (ByteString)
import qualified Data.Text as T
import Data.Text (Text)
import Text.RawString.QQ
import Data.Scientific (floatingOrInteger)

data NumberOrString =
    Numba Integer
  | Stringy Text
  deriving (Eq, Show)

instance FromJSON NumberOrString where
  parseJSON (Number i) =
    case floatingOrInteger i of
      (Left _) ->
        fail "Must be integral number"
      (Right integer) ->
        return $ Numba integer

  parseJSON (String s) = return $ Stringy s
  parseJSON _ =
    fail "NumberOrString must\
         \ be number or string"

-- so it knows what we want to parse
dec :: ByteString
    -> Maybe NumberOrString
dec = decode

eitherDec :: ByteString
          -> Either String NumberOrString
eitherDec = eitherDecode
```

```
main = do
  print $ dec "blah"
```

Now, let's give it a whirl:

```
Prelude> main
Nothing
```

What happened? We can rewrite the code to use `eitherDec` to get a *slightly* more helpful type error:

```
main = do
  print $ dec "blah"
  print $ eitherDec "blah"
```

Then, we can reload the code and try again in the REPL:

```
Prelude> main
Nothing
Left "Error in $: Failed reading:
      not a valid json value"
```

By that means, we are able to get more informative errors from `aeson`. If we want some examples that work, we could try things like the following:

```
Prelude> dec "123"
Just (Numba 123)
Prelude> dec "\"blah\""
Just (Stringy "blah")
```

It's worth getting comfortable with `aeson` even if you don't plan to work with much JSON, because many serialization libraries in Haskell follow a similar API pattern. Play with the example, and see how you need to change the type of `dec` to be able to parse a list of numbers or strings.

24.11 Chapter exercises

1. Write a parser for semantic versions as defined by `http://semver.org/`. After making a working parser, write an `Ord` instance for the `SemVer` type that obeys the specification outlined on the SemVer website:

```
-- Relevant to precedence/ordering,
-- cannot sort numbers like strings.
data NumberOrString =
    NOSS String
  | NOSI Integer

type Major = Integer
type Minor = Integer
type Patch = Integer
type Release = [NumberOrString]
type Metadata = [NumberOrString]

data SemVer =
  SemVer Major Minor Patch Release Metadata

parseSemVer :: Parser SemVer
parseSemVer = undefined
```

Expected results:

```
Prelude> ps = parseString
Prelude> psv = ps parseSemVer mempty
Prelude> psv "2.1.1"
Success (SemVer 2 1 1 [] [])
Prelude> psv "1.0.0-x.7.z.92"
Success (SemVer 1 0 0
         [NOSS "x",
          NOSI 7,
          NOSS "z",
          NOSI 92] [])
```

Some slightly more advanced test cases:

```
Prelude> psv "1.0.0-gamma+002"
Success (SemVer 1 0 0
         [NOSS "gamma"] [NOSI 2])
Prelude> psv "1.0.0-beta+oof.sha.41af286"
Success (SemVer 1 0 0
         [NOSS "beta"]
         [NOSS "oof",
          NOSS "sha",
          NOSS "41af286"])
```

And lastly, the correct total ordering of semantic versions:

```
Prelude> big = SemVer 2 1 1 [] []
Prelude> little = SemVer 2 1 0 [] []
Prelude> big > little
True
```

2. Write a parser for positive integer values. Don't reuse the pre-existing `digit` or `integer` functions, but you can use the rest of the libraries we've shown you so far. You are not expected to write a parsing library from scratch:

```
parseDigit :: Parser Char
parseDigit = undefined

base10Integer :: Parser Integer
base10Integer = undefined
```

Expected results:

```
Prelude> parseString parseDigit mempty "123"
Success '1'
Prelude> parseString parseDigit mempty "abc"
Failure (interactive):1:1: error:
  expected: parseDigit
abc<EOF>
^
Prelude> parseString base10Integer mempty "123abc"
Success 123
```

```
Prelude> parseString base10Integer mempty "abc"
Failure (interactive):1:1: error:
  expected: integer
abc<EOF>
^
```

Hint: Assume you're parsing base-10 numbers. Use arithmetic as a cheap "accumulator" for your final number as you parse each digit left to right.

3. Extend the parser you wrote to handle negative and positive integers. Try writing a new parser in terms of the one you already have in order to do this:

```
Prelude> parseString base10Integer' mempty "-123abc"
Success (-123)
```

4. Write a parser for US/Canada phone numbers with varying formats:

```
-- aka area code
type NumberingPlanArea = Int
type Exchange = Int
type LineNumber = Int

data PhoneNumber =
  PhoneNumber NumberingPlanArea
              Exchange LineNumber
  deriving (Eq, Show)

parsePhone :: Parser PhoneNumber
parsePhone = undefined
```

With the following behavior:

```
Prelude> parseString parsePhone mempty "123-456-7890"
Success (PhoneNumber 123 456 7890)

Prelude> parseString parsePhone mempty "1234567890"
Success (PhoneNumber 123 456 7890)
```

```
Prelude> parseString parsePhone mempty "(123) 456-7890"
Success (PhoneNumber 123 456 7890)

Prelude> parseString parsePhone mempty "1-123-456-7890"
Success (PhoneNumber 123 456 7890)
```

Cf. Wikipedia's article on "National conventions for writing telephone numbers."[11] You are encouraged to adapt the exercise to your locality's conventions, if they are not part of the NNAP scheme.

5. Write a parser for a log file format, and sum the time spent in each activity. Additionally, provide an alternative aggregation of the data that provides average time spent per activity per day. The format supports the use of comments, which your parser will have to ignore. The `#` characters followed by a date mark the beginning of a particular day.

   Log format example:

```
-- wheee a comment

# 2025-02-05
08:00 Breakfast
09:00 Sanitizing moisture collector
11:00 Exercising in high-grav gym
12:00 Lunch
13:00 Programming
17:00 Commuting home in rover
17:30 R&R
19:00 Dinner
21:00 Shower
21:15 Read
22:00 Sleep

# 2025-02-07 -- dates not nececessarily sequential
08:00 Breakfast -- should I try skippin bfast?
```

[11] https://en.wikipedia.org/wiki/National_conventions_for_writing_telephone_numbers

```
09:00 Bumped head, passed out
13:36 Wake up, headache
13:37 Go to medbay
13:40 Patch self up
13:45 Commute home for rest
14:15 Read
21:00 Dinner
21:15 Read
22:00 Sleep
```

You are to derive a reasonable datatype for representing this data yourself. For bonus points, make this bi-directional by making a `Show` representation for the datatype that matches the format you are parsing. Then, write a generator for this data using `QuickCheck`'s `Gen`, and see if you can break your parser with `QuickCheck`.

6. Write a parser for IPv4 addresses:

```
import Data.Word

data IPAddress =
  IPAddress Word32
  deriving (Eq, Ord, Show)
```

A 32-bit word is a 32-bit unsigned int. The lowest value is 0, rather than it being capable of representing negative numbers, but the highest possible value in the same number of bits is twice as high. Note:

```
Prelude> import Data.Int
Prelude> import Data.Word
Prelude> maxBound :: Int32
2147483647
Prelude> maxBound :: Word32
4294967295
Prelude> div 4294967295 2147483647
2
```

`Word32` is an appropriate and compact way to represent IPv4 addresses. You are expected to figure out not only how to parse the typical IP address format but how IP addresses work numerically, insofar as is required to write a working parser. This will require using a search engine, unless you have an appropriate book on internet networking handy.

Example IPv4 addresses and their decimal representations:

```
172.16.254.1 -> 2886794753
204.120.0.15 -> 3430416399
```

7. Same as before, but IPv6:

```
import Data.Word

data IPAddress6 =
  IPAddress6 Word64 Word64
  deriving (Eq, Ord, Show)
```

Example IPv6 addresses and their decimal representations:

```
0:0:0:0:0:ffff:ac10:fe01 -> 281473568538113
0:0:0:0:0:ffff:cc78:f    -> 281474112159759

FE80:0000:0000:0000:0202:B3FF:FE1E:8329 ->
  338288524927261089654163772891438416681

2001:DB8::8:800:200C:417A ->
  42540766411282592856906245548098208122
```

One of the trickier parts about IPv6 will be full vs. collapsed addresses and the abbreviations.[12]

Ensure that you can parse abbreviated variations of the earlier examples, such as:

```
FE80::0202:B3FF:FE1E:8329
2001:DB8::8:800:200C:417A
```

[12]See this Q&A thread about IPv6 abbreviations for more: `http://answers.google.com/answers/threadview/id/770645.html`.

8. Remove the derived `Show` instances from the IPAddress/IPAddress6 types, and write your own `Show` instance for each type that renders in the typical textual format appropriate to each.

9. Write a function that converts between your types for IPAddress and IPAddress6.

10. Write a parser for the DOT language[13] that Graphviz uses to express graphs in plain text.

    We suggest you look at the AST datatype in `haphviz`[14] for ideas on how to represent the graph in a Haskell datatype. If you're feeling especially robust, you can try using fgl.[15]

24.12 Definitions

1. A *parser* parses.

   You read the chapter right?

2. A *parser combinator* combines two or more parsers to produce a new parser. A good example of this is using `<|>` from `Alternative` to produce a new parser from the disjunction of two parser arguments to `<|>`. Or `some`. Or `many`. Or `mappend`. Or `>>`.

3. *Marshalling* is transforming a potentially nonlinear representation of data in memory into a format that can be stored on disk or transmitted over a network socket. Going in the opposite direction is called unmarshalling. Cf. *serialization* and *deserialization*.

4. A *tokenizer* converts text, usually a stream of characters, into more meaningful or "chunkier" structures, such as words, sentences, or symbols—that is, *tokens*. The `lines` and `words` functions you used earlier in this book are like unsophisticated tokenizers.

5. *Lexer*—see *tokenizer*.

[13] `http://www.graphviz.org/doc/info/lang.html`
[14] `http://hackage.haskell.org/package/haphviz`
[15] `http://hackage.haskell.org/package/fgl`

24.13 Follow-up resources

1. Edward Z. Yang. *Parsec "try a <|> b" considered harmful*.
   http://blog.ezyang.com/2014/05/parsec-try-a-or-b-considered-harmful/
2. Bryan O'Sullivan, Don Stewart, and John Goerzen. *Real World Haskell*. Code case study: parsing a binary data format.
   http://book.realworldhaskell.org/read/code-case-study-parsing-a-binary-data-format.html
3. Bryan O'Sullivan, Don Stewart, and John Goerzen. *Real World Haskell*. Using Parsec.
   http://book.realworldhaskell.org/read/using-parsec.html
4. Jakub Arnold. *Parsing CSS with Parsec*.
5. Daan Leijen and Erik Meijer. *Parsec: A practical parser library*.
6. Philip Wadler. *How to Replace Failure by a List of Successes*.
   http://dl.acm.org/citation.cfm?id=5288
7. Edward Kmett. *How to Replace Failure by a Heap of Successes*.
8. Samuel Gélineau. *Two kinds of backtracking*.
9. Josh Haberman. *LL and LR in Context: Why Parsing Tools Are Hard*.
   http://blog.reverberate.org/2013/09/ll-and-lr-in-context-why-parsing-tools.html
10. Dick Grune and Ceriel J. H. Jacobs. *Parsing Techniques: A Practical Guide*. Second Ed.
    https://dickgrune.com/Books/PTAPG_2nd_Edition/
11. School of Haskell. *Parsing JSON with Aeson*.
    https://www.schoolofhaskell.com/school/starting-with-haskell/libraries-and-frameworks/text-manipulation/json
12. Oliver Charles. *24 Days of Hackage: aeson*.

Chapter 25

Composing Types

> The last thing one discovers in composing a work is what to put first.
>
> T. S. Eliot

25.1 Composing types

This chapter and the next are about monad transformers, both the principles behind them and the practicalities of using them. For many programmers, monad transformers are indistinguishable from *magic*, so we want to approach them from both angles and demonstrate that they are both comprehensible via their types and practical in normal programming.

Functors and applicatives are both closed under composition: this means that you can compose two functors (or two applicatives) and return another functor (or applicative, as the case may be). This is not true of monads, however; when you compose two monads, the result is not *necessarily* another monad. We will see this soon.

However, there are times when composing monads is desirable. Different monads allow us to work with different effects. Composing monads allows you to build up computations with multiple effects. By stacking, for example, a `Maybe` monad with an `IO`, you can be performing IO actions while also building up computations that have a possibility of failure, handled by the `Maybe` monad.

A monad transformer is a variant of an ordinary type that takes an additional type argument that is assumed to have a `Monad` instance. For example, `MaybeT` is the transformer variant of the `Maybe` type. The transformer variant of a type gives us a `Monad` instance that binds over both bits of structure. This allows us to compose monads and combine their effects. Getting comfortable with monad transformers is important to becoming proficient in Haskell, so we're going to take it pretty slowly and go step by step. You won't necessarily want to start out early on defining a bunch of transformer stacks yourself, but familiarity with them will help a great deal in using other people's libraries. In this chapter, we will:

- Demonstrate why composing two monads does not give you another monad.
- Examine the `Identity` and `Compose` types.
- Manipulate types until we can make monads compose.
- Meet some common monad transformers.
- Work through an `Identity` crisis.

25.2 Common functions as types

We'll start in a place that may seem a little strange and pointless at first, with newtypes that correspond to some very basic functions. We can construct types that are like those functions, because we have types that can take arguments—that is, type constructors. In particular, we'll be using types that correspond to `id` and the composition operator, `(.)`.

You've seen some of the types we're going to use in the following sections before, but we'll be putting them to some novel uses. The idea here is to use these datatypes as helpers in order to demonstrate the problems with composing monads, and we'll see how these type constructors can also serve as monad transformers, because a monad transformer is a type constructor that takes a monad as an argument.

Identity is boring

You've seen this type in previous chapters, sometimes as a datatype and sometimes as a newtype. We'll construct the type differently this time, as a newtype with a helper function of the sort we saw in `Reader` and `State`:

```
newtype Identity a =
  Identity { runIdentity :: a }
```

We'll be using the newtype in this chapter, because the monad transformer version, `IdentityT`, is usually written as a newtype. The use of the prefixes `run` or `get` indicates that these accessor functions are means of extracting the underlying value from the type. There is no real difference in meaning between `run` and `get`. You'll see these accessor functions often, particularly with utility types such as `Identity` or transformer variants reusing an original type.

A note about newtypes While monad transformer types could be written using the `data` keyword, they are most commonly written as newtypes, and we'll be sticking with that pattern here. They are only newtyped to avoid unnecessary overhead, since newtypes, as you recall, have an underlying representation identical to the type they contain. The important thing is that monad transformers are never sum or product types; they are always a means of wrapping one

extra layer of (monadic) structure around another type, so there is never a reason they couldn't be newtypes. Haskellers have a general tendency to avoid adding additional runtime overhead if they can, so if they can newtype something, they most often will.

Another thing we want to notice about `Identity` is the similarity of the kind of our `Identity` type to the type of the `id` function, although the fidelity of the comparison isn't perfect given the limitations of type-level computation in Haskell:

```
Prelude> :t id
id :: a -> a
Prelude> :k Identity
Identity :: * -> *
```

The kind signature of the type resembles the type signature of the function, which we hope isn't too much of a surprise. Fine, so far—not much new here. Yet.

Compose

We mentioned above that we can also construct a datatype that corresponds to function *composition*.

Here is the `Compose` type. It should look to you much like function composition, but in this case, the `f` and `g` in the type signature represent *type constructors*, not term-level functions:

```
newtype Compose f g a =
  Compose { getCompose :: f (g a) }
  deriving (Eq, Show)
```

So, we have a type constructor that takes three type arguments: `f` and `g` must be type constructors themselves, while `a` will be a concrete type (consider the relationship between type constructors and term-level functions on the one hand, and values and type constants on the other). As we did above, let's look at the kind of `Compose`—note the kinds of the arguments to the type constructor:

```
Compose :: (* -> *) -> (* -> *) -> * -> *
```

Does that remind you of anything?

```
(.) :: (b -> c) -> (a -> b) -> a -> c
```

So, what does that look like in practice? Something like this:

```
Prelude> Compose [Just 1, Nothing]
Compose {getCompose = [Just 1,Nothing]}
Prelude> xs = [Just (1 :: Int), Nothing]
Prelude> :t Compose xs
Compose xs :: Compose [] Maybe Int
```

Given the above value, the type variables get bound accordingly:

```
Compose [Just (1 :: Int), Nothing]

Compose { getCompose :: f (g    a) }

              Compose [] Maybe Int

f ~ []
g ~ Maybe
a ~ Int
```

We have one bit of structure wrapped around another, then a value type (the `a`), because the whole thing still has to be kind `*` in the end.

We've made the point in previous chapters that type constructors are functions. Type constructors can take other type constructors as arguments, too, just as functions can take other functions as arguments. This is what allows us to compose types.

25.3 Two functors sittin' in a tree, L-I-F-T-I-N-G

Let's start with composing functors, using the types we saw above. We know we can lift over `Identity`—you've seen this `Functor` before:

```
instance Functor Identity where
    fmap f (Identity a) = Identity (f a)
```

`Identity` here gives us a sort of vanilla `Functor` that doesn't do anything interesting but captures the essence of what `Functor` is about.

The function gets lifted into the context of the `Identity` type and then mapped over the `a` value.

It turns out we can get a `Functor` instance for `Compose`, too, *if* we require that the `f` and `g` both have `Functor` instances:

```
instance (Functor f, Functor g) =>
         Functor (Compose f g) where
  fmap f (Compose fga) =
    Compose $ (fmap . fmap) f fga
```

Now, the `f` and the `g` both have to be part of the structure that we're lifting over, so they both have to be `Functor` instances, themselves. We need to be able to jump over both of those layers in order to apply to the value that's ultimately inside. We have to `fmap` twice to get to the value inside because of the layered structures.

To return to the example we used above, we have this type:

```
newtype Compose f g a =
  Compose { getCompose :: f (g a) }
  deriving (Eq, Show)
```

```
Compose { getCompose :: f (g    a) }

              Compose [] Maybe Int
```

And if we use our `Functor` instance, we can apply a function to the `Int` value wrapped up in all that structure:

```
Prelude> xs = [Just 1, Nothing]
Prelude> Compose xs
Compose {getCompose = [Just 1,Nothing]}
Prelude> fmap (+1) (Compose xs)
Compose {getCompose = [Just 2,Nothing]}
```

We can generalize this idea to different amounts of structure—for example, one layer of structure less. You may remember this type from a previous chapter:

```
newtype One f a =
  One (f a)
  deriving (Eq, Show)

instance Functor f =>
         Functor (One f) where
  fmap f (One fa) = One $ fmap f fa
```

Or one more layer of structure than `Compose`:

```
newtype Three f g h a =
  Three (f (g (h a)))
  deriving (Eq, Show)

instance (Functor f, Functor g, Functor h)
      => Functor (Three f g h) where
  fmap f (Three fgha) =
    Three $ (fmap . fmap . fmap) f fgha
```

As with the anonymous product `(,)` and the anonymous sum, `Either`, the `Compose` type allows us to express arbitrarily nested types:

```
v :: Compose []
             Maybe
             (Compose Maybe [] Integer)
v = Compose [Just (Compose $ Just [1])]
```

The way to think about this is that the composition of two datatypes that have a `Functor` instance gives rise to a new `Functor` instance. You'll sometimes see people refer to this as functors being *closed under composition*, which means that when you compose two functors, you get another functor.

25.4 Twinplicative

You probably guessed this was our next step in `Compose`-landia. Applicatives, it turns out, are also closed under composition. We can compose two types that have `Applicative` instances and get a new `Applicative` instance. But you're going to write it.

GOTCHA! Exercise time

We provide the instance types for you, as they may help:

```
{-# LANGUAGE InstanceSigs #-}

instance (Applicative f, Applicative g)
      => Applicative (Compose f g) where
  pure :: a -> Compose f g a
  pure = undefined

  (<*>) :: Compose f g (a -> b)
        -> Compose f g a
        -> Compose f g b
  (Compose f) <*> (Compose a) = undefined
```

We mentioned in an earlier chapter that `Applicative` is a weaker algebra than `Monad`, and that sometimes there are benefits to preferring an `Applicative` when you don't need the strength of `Monad`. This is one of those benefits. To compose `Applicatives`, you don't need to do the legwork that a `Monad` requires in order to compose and still have a `Monad`. Oh, yes, right—we still haven't quite made it to monads composing, but we're about to.

25.5 Twonad?

What about `Monad`? There's no problem composing two arbitrary datatypes that have `Monad` instances. We saw this already when we used `Compose` with `Maybe` and list, which both have `Monad` instances defined. However, the result of having done so *does not* give you a `Monad`.

The issue comes down to a lack of information. Both types `Compose` is working with are polymorphic, so when you try to write bind for the `Monad`, you're trying to combine two polymorphic binds into a single, combined bind. This, it turns out, is not possible:

```
{-# LANGUAGE InstanceSigs #-}

-- impossible
instance (Monad f, Monad g)
      => Monad (Compose f g) where
  return = pure

  (>>=) :: Compose f g a
        -> (a -> Compose f g b)
        -> Compose f g b
  (>>=) = ???
```

These are the types we're trying to combine, because `f` and `g` are necessarily both monads with their own `Monad` instances:

```
Monad f => f a -> (a -> f b) -> f b
Monad g => g a -> (a -> g b) -> g b
```

From those, we are trying to write this bind:

```
   (Monad f, Monad g)
=> f (g a) -> (a -> f (g b)) -> f (g b)
```

Or, formulated differently:

```
   (Monad f, Monad g)
=> f (g (f (g a))) -> f (g a)
```

And this is not possible. There's no good way to `join` that final `f` and `g`. It's a great exercise to try to make it work, because the barriers you'll run into are instructive in their own right.[1]

No free burrito lunches

Since getting another `Monad` given the composition of two *arbitrary* types that have a `Monad` instance is impossible, what can we do to get a `Monad` instance for combinations of types? The answer is *monad transformers*. We'll get to those after a little break for some exercises.

[1] You can also read *Composing monads* by Mark P. Jones and Luc Duponcheel to see why it's not possible: `http://web.cecs.pdx.edu/~mpj/pubs/RR-1004.pdf`.

25.6 Exercises: Compose instances

1. Write the `Compose Foldable` instance.

   The `foldMap = undefined` bit is a hint to make it easier and look more like what you've seen, already:

   ```
   instance (Foldable f, Foldable g) =>
            Foldable (Compose f g) where
       foldMap = undefined
   ```

2. Write the `Compose Traversable` instance:

   ```
   instance (Traversable f, Traversable g) =>
            Traversable (Compose f g) where
       traverse = undefined
   ```

And now for something completely different

This has nothing to do with anything else in this chapter, but it makes for a fun exercise:

```
class Bifunctor p where
    {-# MINIMAL bimap | first, second #-}

    bimap :: (a -> b)
          -> (c -> d)
          -> p a c
          -> p b d
    bimap f g = first f . second g

    first :: (a -> b) -> p a c -> p b c
    first f = bimap f id

    second :: (b -> c) -> p a b -> p a c
    second = bimap id
```

It's a functor that can map over two type arguments instead of one. Write `Bifunctor` instances for the following types. The less you think, the easier it'll be:

1. `data Deux a b = Deux a b`

2. ```
   data Const a b = Const a
   ```

3. ```
   data Drei a b c = Drei a b c
   ```

4. ```
   data SuperDrei a b c = SuperDrei a b
   ```

5. ```
   data SemiDrei a b c = SemiDrei a
   ```

6. ```
   data Quadriceps a b c d =
       Quadzzz a b c d
   ```

7. ```
   data Either a b =
       Left a
     | Right b
   ```

25.7 Monad transformers

We've now seen what the problem with `Monad` is: you can put two together, but you don't get a new `Monad` instance out of it. When we need to get a new `Monad` instance, we need a monad transformer. It's not magic; the answer is in the types.

We said above that a monad transformer is a type constructor that takes a `Monad` as an argument and returns a `Monad` as a result. We also noted that the fundamental problem with composing two monads lies in the impossibility of joining two unknown monads. In order to make that `join` happen, we need to reduce the polymorphism and get concrete information about one of the monads that we're working with. The other monad remains polymorphic as a variable type argument to our type constructor. Transformers help you make a monad out of multiple types—however many there are—that each have a `Monad` instance, by wrapping these existing monads, each of which provides a bit of the functionality you want.

The types are tricky here, so we're going to be walking through the process of writing monad transformers slowly. Parts of what follows may seem tedious, so work through it as slowly or quickly as you need to.

Monadic stacking

`Applicative` allows us to apply functions of more than one argument in the presence of functorial structure, enabling us to cope with this transition:

```
-- from this:
fmap (+1) (Just 1)

-- to this:
   (,,)
<$> Just 1
<*> Just "lol"
<*> Just [1, 2]
```

Sometimes, we want a `>>=` that can address more than one `Monad` at once. You'll often see this in applications that have multiple things going on, such as a web app where combining `Reader` and `IO` is common. You want `IO`, so you can perform effectful actions like talking to a database and also `Reader` for the database connection(s) and/or HTTP request context. Sometimes, you may even want multiple `Reader` types (app-specific data vs. what the framework provides by default), although usually there's a way to add only the data you want to a product type of a single `Reader`.

So the question becomes, how do we get *one big bind* over a type like the following, where the `Monad` instances involved are `IO`, `Reader`, and `[]`?

```
IO (Reader String [a])
```

Doing it badly

We could make one-off types for each combination, but this will get tiresome quickly. For example:

```
newtype MaybeIO a =
  MaybeIO { runMaybeIO :: IO (Maybe a) }

newtype MaybeList a =
  MaybeList { runMaybeList :: [Maybe a] }
```

We don't need to resort to this, however. We can instead get a `Monad` for *two* types, as long as we know what *one* of the types is. Transformers are a means of avoiding making a one-off `Monad` for every possible combination of types.

25.8 IdentityT

Much as `Identity` helps show off the most basic essence of `Functor`, `Applicative`, and `Monad`, `IdentityT` is going to help you begin to understand monad transformers. Using this type that doesn't have a lot of interesting stuff going on with it will help keep us focused on the types and the important fundamentals of transformers. What we see here will be applicable to other transformers, as well, but types like `Maybe` and list introduce other possibilities (failure cases, empty lists) that complicate things a bit.

First, let's compare the `Identity` type you've seen up to this point and our new `IdentityT` datatype. First, here's the base type:

```
newtype Identity a =
  Identity { runIdentity :: a }
  deriving (Eq, Show)
```

And now, here's the identity monad transformer, which serves only to specify that additional structure should exist:

```
newtype IdentityT f a =
  IdentityT { runIdentityT :: f a }
  deriving (Eq, Show)
```

What changes here is that we add an extra type argument. Then, we want `Functor` instances for both `Identity` and `IdentityT`:

```
instance Functor Identity where
  fmap f (Identity a) = Identity (f a)

instance (Functor m)
      => Functor (IdentityT m) where
  fmap f (IdentityT fa) =
    IdentityT (fmap f fa)
```

The `IdentityT` instance should look similar to the `Functor` instance for the `One` datatype above—the `fa` argument is the value inside the `IdentityT` with the (untouchable) structure wrapped around it. All we know about that additional layer of structure wrapped around the `a` value is that it is a `Functor`.

We also want `Applicative` instances for each:

```
instance Applicative Identity where
  pure = Identity

  (Identity f) <*> (Identity a) =
    Identity (f a)

instance (Applicative m)
      => Applicative (IdentityT m) where
    pure x = IdentityT (pure x)

    (IdentityT fab) <*> (IdentityT fa) =
      IdentityT (fab <*> fa)
```

The `Identity` instance should be familiar. In the `IdentityT` instance, the `fab` variable represents the `f (a -> b)` that is the first argument of `<*>`. Since this can rely on the `Applicative` instance for `m` to handle that bit, this instance defines how to applicatively apply in the presence of that outer `IdentityT` layer.

Finally, we want some `Monad` instances:

```
instance Monad Identity where
  return = pure

  (Identity a) >>= f = f a

instance (Monad m)
      => Monad (IdentityT m) where
    return = pure
    (IdentityT ma) >>= f =
      IdentityT $ ma >>= runIdentityT . f
```

The `Monad` instance is tricky, so we're going to do a few things to break it down. Keep in mind that `Monad` is where we must use concrete type information from `IdentityT` in order to make the types fit.

The bind breakdown

We'll start with a closer look at the instance, as written above:

```
instance (Monad m)
      => Monad (IdentityT m) where
    return = pure
    (IdentityT ma) >>=  f  =
--   [     1     ] [2] [3]
       IdentityT $ ma
--         [8]     [4]
       >>= runIdentityT . f
--     [5]      [7]      [6]
```

1. First, we pattern match or unpack the `m a` value of `IdentityT m a` via the data constructor. This operation has the type `IdentityT m a -> m a`, and the type of `ma` is `m a`. This nomenclature doesn't mean anything beyond mnemonic signaling, but it is intended to be helpful.

2. The type of the bind we are implementing is the following:

   ```
   (>>=) :: IdentityT m a
         -> (a -> IdentityT m b)
         -> IdentityT m b
   ```

   This is the instance we are defining.

3. The function we're binding over is `IdentityT m a`. It has the type `(a -> IdentityT m b)`.

4. Here, `ma` is the same one we unpack out of the `IdentityT` data constructor and has the type `m a`. Removed from its `IdentityT` context, this is the `m a` that the bind takes as its first argument.

5. This is a *different* bind! The first bind is the bind we're trying to implement; this bind is its definition or implementation. We're now using the `Monad` we ask for in the instance declaration with the constraint `Monad m =>`. This will have the type:

   ```
   (>>=) :: m a -> (a -> m b) -> m b
   ```

This is with respect to the `m` in the type `IdentityT m a`, not the class of `Monad` instances, in general. In other words, since we have already unpacked the `IdentityT` bit and, in a sense, gotten it out of the way, this bind will be the bind for the type `m` in the type `IdentityT m`. We don't know what `Monad` that is yet, and we don't need to. Since it has the `Monad` type class constraint on that variable, we know it already has a `Monad` instance defined for it, and this second bind will be the bind defined for that type. All we're doing here is defining how to use that bind in the presence of the additional `IdentityT` structure.

6. This is the same `f` that is an argument to the `Monad` instance we are defining, of type `(a -> IdentityT m b)`.

7. We need `runIdentityT`, because `f` returns `IdentityT m b`, but the `>>=` for the `Monad m =>` constraint has the type `m a -> (a -> m b) -> m b`. It'll end up trying to join `m (IdentityT m b)`, which won't work, because `m` and `IdentityT m` are not the same type. We use `runIdentityT` to unpack the value. Doing this has the type `IdentityT m b -> m b`, and the composition `runIdentityT . f` in this context has the type `a -> m b`. You can use `undefined` in GHCi to demonstrate this for yourself:

```
Prelude> :{
*Main| let f :: (a -> IdentityT m b)
*Main|     f = undefined
*Main| :}
Prelude> :t f
f :: a -> IdentityT m b
Prelude> :t runIdentityT
runIdentityT :: IdentityT f a -> f a
Prelude> :t (runIdentityT . f)
(runIdentityT . f) :: a1 -> f a
```

OK, the type variables don't have the same name, but you can see how `a1 -> f a` and `a -> m b` are the same type.

8. To satisfy the type of the outer bind we are implementing for the `Monad` of `IdentityT m`, which expects a final result of the type

`IdentityT m b`, we must take the `m b` that the expression `ma >>= runIdentityT . f` returns and *repack* it into `IdentityT`. Note:

```
Prelude> :t IdentityT
IdentityT :: f a -> IdentityT f a
Prelude> :t runIdentityT
runIdentityT :: IdentityT f a -> f a
```

Now we have a bind we can use with `IdentityT` and some other `Monad`—in this example, a list:

```
Prelude> sumR = return . (+1)
Prelude> IdentityT [1, 2, 3] >>= sumR
IdentityT {runIdentityT = [2,3,4]}
```

Implementing the bind, step by step

We're going to backtrack and go through implementing that bind step by step. The goal here is to demystify what we've done and enable you to write your own instances for whatever monad transformer you might need to implement yourself. We'll go ahead and start back at the beginning, but with `InstanceSigs` turned on so we can see the type:

```
{-# LANGUAGE InstanceSigs #-}

instance (Monad m)
      => Monad (IdentityT m) where
    return = pure

    (>>=) :: IdentityT m a
          -> (a -> IdentityT m b)
          -> IdentityT m b
    (IdentityT ma) >>= f =
      undefined
```

Let's leave the `undefined` as our final return expression, then use `let` bindings and contradiction to see the types of our attempts at making a `Monad` instance. We're going to use the bottom value (`undefined`) to defer the parts of the proof we're obligated to produce until we're

ready. First, let's get a `let` binding in place and see it load, even if the code doesn't work:

```
    (>>=) :: IdentityT m a
          -> (a -> IdentityT m b)
          -> IdentityT m b
    (IdentityT ma) >>= f =
      let aimb = ma >>= f
      in undefined
```

We're using `aimb` as a mnemonic for the parts of the whole thing that we're trying to implement.

Here we get an error:

```
Couldn't match type ‘m’ with ‘IdentityT m’
```

That type error isn't the most helpful thing in the world. It's hard to know what's wrong from reading it. So, we'll poke at this a bit in order to get a more helpful type error.

First, we'll do something we *know* should work. We'll use `fmap` instead. Because that will type check (but not give us the same result as `>>=`), we need to do something to give the compiler a chance to contradict us and tell us the real type. We force that type error by asserting a fully polymorphic type for `aimb`:

```
(>>=) :: IdentityT m a
      -> (a -> IdentityT m b)
      -> IdentityT m b
(IdentityT ma) >>= f =
  let aimb :: a
      aimb = fmap f ma
  in undefined
```

The type we assert for `aimb` is impossible; we've said it could be any type, and it can't. The only thing that can have that type is bottom, as bottom inhabits all types.

Conveniently, GHC will let us know what `aimb` is:

```
Couldn't match expected type ‘a1’
  with actual type ‘m (IdentityT m b)’
```

With the current implementation, `aimb` has the type `m (IdentityT m b)`. Now, we can see the real problem: there is an `IdentityT` layer in between the two bits of `m` that we need to join in order to have a monad.

Here's a breakdown:

```
(>>=) :: IdentityT m a
      -> (a -> IdentityT m b)
      -> IdentityT m b
```

The pattern match on `IdentityT` comes from having lifted over it:

```
(a -> IdentityT m b)
```

The problem is, we use `>>=` over `m a` and get `m (IdentityT m b)`.

It doesn't type check, because `>>=` merges structure of the same type after lifting (remember: it's `fmap` composed with `join` under the hood). Had our type been `m (m b)` after binding `f` over `ma`, it would have worked fine. As it is, we need to find a way to get the two bits of `m` together without an intervening `IdentityT` layer.

We're going to continue with separate `fmap` and `join` functions, instead of using `>>=`, because it makes the step by step manipulation of structure easier to see. How do we get rid of the `IdentityT` in the middle of the two `m` structures? Well, we know `m` is a `Monad`, which means it's also a `Functor`. So, we can use `runIdentityT` to get rid of the `IdentityT` structure in the middle of the stack of types:

```
-- Change m (IdentityT m b)
-- into m (m b)

-- Note:
runIdentityT :: IdentityT f  a  ->     f a
fmap runIdentityT :: Functor f
         => f (IdentityT f1 a) -> f (f1 a)
```

```
(>>=) :: IdentityT m a
      -> (a -> IdentityT m b)
      -> IdentityT m b
(IdentityT ma) >>= f =
  let aimb :: a
      aimb = fmap runIdentityT (fmap f ma)
  in undefined
```

And when we load this code, we get an encouraging type error:

```
Couldn't match expected type ‘a1’
  with actual type ‘m (m b)’
```

It's telling us we have achieved the type `m (m b)`, so now we know how to get where we want. The `a1` here is the `a` we assign to `aimb`, but it's telling us that our actual type is not what we assert but this other type. Thus, we have discovered what our actual type is, which gives us a clue about how to fix it.

We'll use `join` from `Control.Monad` to merge the nested `m` structure:

```
(>>=) :: IdentityT m a
      -> (a -> IdentityT m b)
      -> IdentityT m b
(IdentityT ma) >>= f =
  let aimb :: a
      aimb =
       join (fmap runIdentityT (fmap f ma))
  in undefined
```

And when we load it, the compiler tells us we finally have an `m b` that we can return:

```
Couldn't match expected type ‘a1’
  with actual type ‘m b’
```

In fact, before we begin cleaning up our code, we can verify this is the case real quick:

```
(>>=) :: IdentityT m a
      -> (a -> IdentityT m b)
      -> IdentityT m b
(IdentityT ma) >>= f =
  let aimb =
       join (fmap runIdentityT (fmap f ma))
  in aimb
```

We remove the type declaration for `aimb` and also change the `in undefined`. But we know that `aimb` has the actual type `m b`, so this won't work. Why? If we take a look at the type error:

```
Couldn't match type ‘m’ with ‘IdentityT m’
```

The `>>=` we are implementing has a final result of type `IdentityT m b`, so the type of `aimb` doesn't match it yet. We need to wrap `m b` in `IdentityT` in order to make it type check:

```
-- Remember:
IdentityT :: f a -> IdentityT f a

instance (Monad m)
      => Monad (IdentityT m) where
    return = pure

    (>>=) :: IdentityT m a
          -> (a -> IdentityT m b)
          -> IdentityT m b
    (IdentityT ma) >>= f =
      let aimb =
           join (fmap runIdentityT
                  (fmap f ma))
      in IdentityT aimb
```

This should compile. We rewrap `m b` back in the `IdentityT` type, and we should be good to go.

Refactoring

Now that we have something that works, let's refactor. We'd like to improve our implementation of `>>=`. Taking things one step at a time is usually more successful than trying to rewrite all at once, especially once you have a baseline version that you know should work. How can we improve this line?

```
IdentityT $
join (fmap runIdentityT (fmap f ma))
```

Well, one of the `Functor` laws tells us something about performing an `fmap` twice:

```
-- Functor law:
fmap (f . g)  ==  fmap f . fmap g
```

Indeed! So we can change that line to the following, and it should be identical:

```
IdentityT $
join (fmap (runIdentityT . f) ma)
```

It seems suspicious that we're using `fmap` and also `join` on the result of having `fmap`ped the two functions we compose. Isn't `join` composed with `fmap` just `>>=`?

```
x >>= f = join (fmap f x)
```

Accordingly, we can change our `Monad` instance to the following:

```
instance (Monad m)
      => Monad (IdentityT m) where
    return = pure

    (>>=) :: IdentityT m a
          -> (a -> IdentityT m b)
          -> IdentityT m b
    (IdentityT ma) >>= f =
      IdentityT $ ma >>= runIdentityT . f
```

And that should still work! We have a type constructor now (`IdentityT`) that takes a monad as an argument and returns a monad as a result.

This implementation can be written in other ways. In the `transformers` library, for example, it's written like this:

```
m >>= k =
  IdentityT $ runIdentityT . k
  =<< runIdentityT m
```

Take a moment and work out for yourself how that is functionally equivalent to our implementation.

The essential extra of monad transformers

It may not seem like it, but the `IdentityT` monad transformer captures the essence of transformers, generally. We only embarked on this quest, because we couldn't be guaranteed a `Monad` instance given the composition of two types. Given that, we know having `Functor`, `Applicative`, and `Monad` at our disposal isn't enough to make that new `Monad` instance. So what is novel in the following code?

```
(>>=) :: IdentityT m a
      -> (a -> IdentityT m b)
      -> IdentityT m b
(IdentityT ma) >>= f =
  IdentityT $ ma >>= runIdentityT . f
```

It isn't the pattern match on `IdentityT`; we get that from the `Functor` anyway:

```
-- Not this
(IdentityT ma) ...
```

It isn't the ability to bind functions with `>>=` over the `ma` value of type `m a`, since we get that from the `Monad` constraint on `m`, anyway:

```
-- Not this
... ma >>= ...
```

We need to know one of the types concretely so that we can use `runIdentityT` (essentially `fmap`ping a *fold* of the `IdentityT` structure) and then repack the value in `IdentityT`:

```
-- We need to know IdentityT
-- concretely to do this
IdentityT ... runIdentityT ...
```

As you'll recall, until we use `runIdentityT`, we can't get the types to fit, because `IdentityT` is wedged in the middle of two bits of `m`. It turns out to be impossible to fix that using only `Functor`, `Applicative`, and `Monad`. This is an example of why we can't make a `Monad` instance for the `Compose` type, but we *can* make a transformer type like `IdentityT` where we leverage information specific to the type and combine it with any other type that has a `Monad` instance. In general, in order to make the types fit, we'll need some way to fold and reconstruct the type for which we do have concrete information.

25.9 Finding a pattern

Transformers are bearers of single-type concrete information that let you create ever-bigger monads, in a sense. Nesting such as:

```
(Monad m) => m (m a)
```

Is addressed by `join` already. We use transformers when we want a `>>=` operation over `f` and `g` of different types (but both have `Monad` instances). You have to create new types called monad transformers and write `Monad` instances for those types to have a way of dealing with the extra structure generated.

The general pattern is this: You want to compose two polymorphic types, `f` and `g`, that each have a `Monad` instance. But you'll end up with this pattern:

```
f (g (f b))
```

`Monad`'s bind can't join those types, not with that intervening `g`. So you need to get to this:

```
f (f b)
```

You won't be able to unless you have some way of *folding* the `g` in the middle. You can't do that with `Monad`. The essence of `Monad` is `join`, but here you have only one bit of `g` structure, not `g (g ...)`, so that's not enough. The straightforward thing to do is to make `g` concrete. With concrete type information for the inner bit of structure, we can fold out the `g` and get on with it. The good news is that transformers don't require `f` to be concrete; `f` can remain polymorphic, so long as it has a `Monad` instance, and therefore we only need to write a transformer once for each type.

We can see this pattern with `IdentityT`, as well. You may recall this step in our process of writing the `Monad` instance for `IdentityT`:

```
(IdentityT ma) >>= f =
  let aimb :: m (IdentityT m b)
      aimb = fmap f ma
```

We have *something* that will type check, but it's not exactly in the shape we would like. Of course, the underlying type, once we throw away the `IdentityT` data constructor, is `m (m b)`, which will suit us just fine, but we have to fold out the `IdentityT` before we can use the `join` from `Monad m => m`. That leads us to the next step:

```
let aimb :: m (m b)
    aimb = fmap runIdentityT (fmap f ma)
```

Now, we finally have something we can join, because we lift the record accessor for `IdentityT` over the `m`! Since `IdentityT` is so simple, the record accessor is sufficient to fold up the structure. From there, the following transitions become easy:

```
m (m b) -> m b -> IdentityT m b
```

The final type is what our definition of `>>=` for `IdentityT` must result in.

The basic pattern that many monad transformers are enabling us to cope with entails the following type transitions, where `m` is the polymorphic, outer structure and `T` is some concrete type the transformer is for. For example, in the above, `T` would be `IdentityT`:

```
   m (T m b)
-> m (m b)
-> m b
-> T m b
```

Don't consider this a hard and fast rule for what types you'll encounter in implementing transformers, but rather some intuition for why transformers are necessary in the first place.

Chapter 26

Monad Transformers

> I do not say such things except insofar as I consider this to permit some transformation of things. Everything I do, I do in order that it may be of use.
>
> Michel Foucault

26.1 Monad transformers

The last chapter demonstrated why we need monad transformers and the basic type manipulation that's going on to make that bit of magic happen. Monad transformers are important in a lot of everyday Haskell code, though, so we want to dive deeper and make sure we have provided a good understanding of how to use them in practice. Even after you know how to write all the transformer instances, managing stacks of transformers in an application can be tricky. The goal of this chapter is to get comfortable with it.

In this chapter, we will:

- Work through more monad transformer types and instances.
- Look at the ordering and wrapping of monad transformer stacks.
- Lift, lift, lift, and lift some more.

26.2 MaybeT

In the last chapter, we worked through an extended breakdown of the `IdentityT` transformer. `IdentityT` is, as you might imagine, not the most useful of the monad transformers, although it is not without practical applications (more on this later). As we've seen, though, the `Maybe Monad` can be useful, and so it is that the transformer variant, `MaybeT`, finds its way into the pantheon of important transformers.

The `MaybeT` transformer is a bit more complex than `IdentityT`. If you worked through all the exercises of the previous chapter, then this section will not be too surprising, as it will rely on things you've seen with `IdentityT` and the `Compose` type already. However, to ensure that transformers are thoroughly demystified for you, it's worth working through them carefully.

We begin with the newtype for our transformer:

```
newtype MaybeT m a =
  MaybeT { runMaybeT :: m (Maybe a) }
```

The structure of our `MaybeT` type and the `Compose` type are similar, so we can reuse the basic patterns of the `Compose` type for the `Functor` and `Applicative` instances:

```
instance (Functor f, Functor g)
      => Functor (Compose f g) where
  fmap f (Compose fga) =
    Compose $ (fmap . fmap) f fga
```

Compare this to the instance for `MaybeT`:

```
instance (Functor m)
      => Functor (MaybeT m) where
  fmap f (MaybeT ma) =
    MaybeT $ (fmap . fmap) f ma
```

We don't need to do anything different for the `Functor` instance, because transformers are needed for the `Monad`, not the `Functor`.

Spoiler alert!

If you haven't yet written the `Applicative` instance for `Compose` from the previous chapter, you may want to stop right here.

We'll start with what might seem like an obvious way to write the `MaybeT Applicative` and find out why it doesn't work. This does not compile:

```
instance (Applicative m)
      => Applicative (MaybeT m) where
  pure x = MaybeT (pure (pure x))

  (MaybeT fab) <*> (MaybeT mma) =
    MaybeT $ fab <*> mma
```

The `fab` represents the function `m (Maybe (a -> b))` and the `mma` represents the `m (Maybe a)`.

You'll get this error if you try it:

```
Couldn't match type ‘Maybe (a -> b)’
with ‘Maybe a -> Maybe b’
```

Here is the `Applicative` instance for `Compose` as a comparison with the `MaybeT` instance we're trying to write:

```
instance (Applicative f, Applicative g)
      => Applicative (Compose f g) where
    pure x = Compose (pure (pure x))
    Compose f <*> Compose x =
      Compose ((<*>) <$> f <*> x)
```

Let's break this down a bit in case you felt confused when you wrote this for the last chapter's exercise. Because you did that exercise… right?

The idea here is that we have to lift an `Applicative` apply over the outer structure `f` in order to transform the `g (a -> b)` into `g a -> g b`, so that we can leverage the `Applicative` instance for `f`. We can stretch this idea a bit and use concrete types:

```
innerMost
  :: [Maybe (Identity (a -> b))]
  -> [Maybe (Identity a -> Identity b)]
innerMost = (fmap . fmap) (<*>)

second'
  :: [Maybe (Identity a -> Identity b)]
  -> [ Maybe (Identity a)
    -> Maybe (Identity b) ]
second' = fmap (<*>)

final'
  :: [ Maybe (Identity a)
    -> Maybe (Identity b) ]
  -> [Maybe (Identity a)]
  -> [Maybe (Identity b)]
final' = (<*>)
```

The function that could be the `Applicative` instance for such a hypothetical type would look like this:

```
lmiApply :: [Maybe (Identity (a -> b))]
         -> [Maybe (Identity a)]
         -> [Maybe (Identity b)]
lmiApply f x =
  final' (second' (innerMost f)) x
```

The `Applicative` instance for our `MaybeT` type will employ this same idea, because applicatives are closed under composition, as we noted in the last chapter. We only need to do something different from the `Compose` instances once we get to `Monad`.

So, we took the long way around to this:

```
instance (Applicative m)
      => Applicative (MaybeT m) where
  pure x = MaybeT (pure (pure x))

  (MaybeT fab) <*> (MaybeT mma) =
    MaybeT $ (<*>) <$> fab <*> mma
```

MaybeT Monad instance

At last, on to the `Monad` instance! Note that we've given some of the intermediate types:

```
instance (Monad m)
      => Monad (MaybeT m) where
    return = pure

    (>>=) :: MaybeT m a
          -> (a -> MaybeT m b)
          -> MaybeT m b

    (MaybeT ma) >>= f =
--          [2]    [3]
      MaybeT $ do
--    [ 1 ]

        -- ma :: m (Maybe a)
        -- v :: Maybe a
        v <- ma
--        [4]
        case v of
--          [5]
            Nothing -> return Nothing
--          [          6          ]
```

```
            Just y -> runMaybeT (f y)
            --           [7]     [8]
-- y :: a
-- f :: a -> MaybeT m b
-- f y :: MaybeT m b
-- runMaybeT (f y) :: m (Maybe b)
```

Explaining it step by step:

1. We have to return a `MaybeT` value at the end, so the `do` block has the `MaybeT` data constructor in front of it. This means the final value of our `do` block expression must be of type `m (Maybe b)` in order to type check, because our goal is to go from `MaybeT m a` to `MaybeT m b`.

2. The first argument to bind here is `MaybeT m a`. We unbundle that from `MaybeT` by pattern matching on the `MaybeT` newtype data constructor.

3. The second argument to bind is `(a -> MaybeT m b)`.

4. In the definition of `MaybeT`, notice something:

   ```
   newtype MaybeT m a =
     MaybeT { runMaybeT :: m (Maybe a) }
   --                      ^---------^
   ```

   It's a `Maybe` value wrapped in *some other type*, for which all we know is that it has a `Monad` instance. Accordingly, we begin in our `do` block by using the left arrow bind syntax. This gives us a reference to the hypothetical `Maybe` value inside of the `m` structure, which is unknown.

5. Since using `<-` to bind `Maybe a` out of `m (Maybe a)` leaves us with a `Maybe` value, we use a case expression on the `Maybe` value.

6. If we get `Nothing`, we kick `Nothing` back out, but we have to embed it in the `m` structure. We don't know what `m` is, but being a `Monad` (and thus also an `Applicative`) means we can use `return` (or `pure`) to perform that embedding.

7. If we get `Just`, we now have a value of type `a` that we can pass to our function `f` of type `a -> MaybeT m b`.

8. We have to fold the `m (Maybe b)` value out of the `MaybeT`, since the `MaybeT` constructor is already wrapped around the whole `do` block, and then we're done.

Don't be afraid to get a pen and paper and work all that out until you understand how things are happening before you move on.

26.3 EitherT

Just as `Maybe` has a transformer variant in the form of `MaybeT`, we can make a transformer variant of `Either`. We'll call it `EitherT`. Your task is to implement the instances for the transformer variant:

```
newtype EitherT e m a =
  EitherT { runEitherT :: m (Either e a) }
```

Exercises: EitherT

1. Write the `Functor` instance for `EitherT`:

```
instance Functor m
      => Functor (EitherT e m) where
  fmap = undefined
```

2. Write the `Applicative` instance for `EitherT`:

```
instance Applicative m
      => Applicative (EitherT e m) where
  pure = undefined

  f <*> a = undefined
```

3. Write the `Monad` instance for `EitherT`:

```
instance Monad m
      => Monad (EitherT e m) where
  return = pure

  v >>= f = undefined
```

4. Write the `swapEitherT` helper function for `EitherT`:

```
-- transformer version of swapEither
swapEitherT :: (Functor m)
            => EitherT e m a
            -> EitherT a m e
swapEitherT = undefined
```

Hint: write `swapEither` first, then write `swapEitherT` in terms of the former.

5. Write the transformer variant of the `either` catamorphism:

```
eitherT :: Monad m =>
           (a -> m c)
        -> (b -> m c)
        -> EitherT a m b
        -> m c
eitherT = undefined
```

26.4 ReaderT

`ReaderT` is one of the most commonly used transformers in conventional Haskell applications. It is like `Reader`, except in the transformer variant, we're generating additional structure in the return type of the function:

```
newtype ReaderT r m a =
  ReaderT { runReaderT :: r -> m a }
```

The value inside the `ReaderT` is a *function*. Type constructors such as `Maybe` are also functions in some sense, but we have to handle this case a bit differently. The first argument to the function inside `ReaderT` is part of the structure we'll have to bind over.

This time, we're going to give you the instances. If you want to try writing them yourself, *do not read on*!

```
instance (Functor m)
      => Functor (ReaderT r m) where
  fmap f (ReaderT rma) =
    ReaderT $ (fmap . fmap) f rma
```

```
instance (Applicative m)
      => Applicative (ReaderT r m) where
  pure a = ReaderT (pure (pure a))

  (ReaderT fmab) <*> (ReaderT rma) =
    ReaderT $ (<*>) <$> fmab <*> rma

instance (Monad m)
      => Monad (ReaderT r m) where
  return = pure

  (>>=) :: ReaderT r m a
        -> (a -> ReaderT r m b)
        -> ReaderT r m b
  (ReaderT rma) >>= f =
    ReaderT $ \r -> do
--            [1]
      a <- rma r
--     [3] [ 2 ]
      runReaderT (f a)  r
--       [5]     [ 4 ] [6]
```

1. Again, the type of the value in a `ReaderT` must be a function, so the act of binding a function over a `ReaderT` must itself be a function awaiting an argument of type `r`, which we've chosen to name `r` as a convenience in our terms. Also note that we're repacking our lambda inside the `ReaderT` data constructor.

2. We pattern match the `r -> m a` (represented in our terms by `rma`) out of the `ReaderT` data constructor. Now, we're applying it to the `r` that we're expecting in the body of the anonymous lambda.

3. The result of applying `r -> m a` to a value of type `r` is `m a`. We need a value of type `a` in order to apply our `a -> ReaderT r m b` function. To be able to write code in terms of that hypothetical `a`, we bind (`<-`) the `a` out of the `m` structure. We bind that value to the name `a` as a mnemonic to remember the type.

4. Applying `f`, which has the type `a -> ReaderT r m b`, to the value `a` results in a value of type `ReaderT r m b`.

5. We unpack the `r -> m b` out of the `ReaderT` structure.

6. Finally, we apply the resulting `r -> m b` to the `r` we have at the beginning of our lambda, that eventual argument that `Reader` abstracts for us. We have to return `m b` as the final expression in this anonymous lambda or the function is not valid. To be valid, it must be of type `r -> m b`, which expresses the constraint that if it is applied to an argument of type `r`, it must produce a value of type `m b`.

No exercises this time. You deserve a break.

26.5 StateT

Similar to `Reader` and `ReaderT`, `StateT` is `State` but with additional monadic structure wrapped around the result. `StateT` is somewhat more useful and common than the `State Monad` you saw earlier. Like `ReaderT`, its value is a function:

```
newtype StateT s m a =
  StateT { runStateT :: s -> m (a,s) }
```

Exercises: StateT

If you're familiar with the distinction, you'll be implementing the strict variant of `StateT` here. To make the strict variant, you don't have to do anything special. Write the most obvious thing that could work. The lazy (lazier, anyway) variant is the one that involves adding a bit extra. We'll explain the difference in the next chapter, on non-strictness.

1. You'll have to do the `Functor` and `Applicative` instances first, because there aren't any `Functor` and `Applicative` instances ready to go for the type `Monad m => s -> m (a, s)`:

```
instance (Functor m)
      => Functor (StateT s m) where
    fmap f m = undefined
```

2. As with `Functor`, you can't cheat and reuse an underlying `Applicative` instance, so you'll have to do the work with the `s -> m (a, s)` type yourself:

```
instance (Monad m)
      => Applicative (StateT s m) where
    pure  = undefined
    (<*>) = undefined
```

   Also, note that the constraint on `m` is *not* `Applicative` as you expect, but rather `Monad`. This is because you can't express the order-dependent computation you'd expect the `StateT Applicative` to have without having a `Monad` for `m`. In essence, the issue is that without `Monad`, you're feeding the initial state to each computation in `StateT` rather than threading it through as you go. This is a general pattern contrasting `Applicative` and `Monad` and is worth contemplating.

3. The `Monad` instance should look *fairly* similar to the `Monad` instance you wrote for `ReaderT`:

```
instance (Monad m)
      => Monad (StateT s m) where
  return = pure

  sma >>= f = undefined
```

ReaderT, WriterT, StateT

We'd like to point something out about these three types:

```
newtype Reader r a =
  Reader { runReader :: r -> a }

newtype Writer w a =
  Writer { runWriter :: (a, w) }

newtype State s a =
  State { runState :: s -> (a, s) }
```

And their transformer variants:

```
newtype ReaderT r m a =
  ReaderT { runReaderT :: r -> m a }

newtype WriterT w m a =
  WriterT { runWriterT :: m (a, w) }

newtype StateT s m a =
  StateT { runStateT :: s -> m (a, s) }
```

You're already familiar with `Reader` and `State`. We haven't shown you `Writer` or `WriterT` up to this point because, quite frankly, you shouldn't use them. We'll explain why not in a section later in this chapter.

For the purposes of the progression we're trying to demonstrate here, it suffices to know that the `Writer` `Applicative` and `Monad` work by combining the `w` values monoidally. With that in mind, what we can see is that `Reader` lets us talk about values we need, `Writer` lets us deal with values we can emit and combine (but not read), and `State` lets us both read and write values in any manner we desire—including monoidally, like `Writer`. This is one reason you needn't bother with `Writer`, since `State` can replace it anyway. That's why you don't *need* `Writer`. We'll discuss more about why you don't *want* `Writer` later.

In fact, there is a type in the `transformers` library that combines `Reader`, `Writer`, and `State` into one big type:

```
newtype RWST r w s m a =
  RWST { runRWST :: r -> s -> m (a, s, w) }
```

Because of the `Writer` component, you probably wouldn't want to use this type in most applications either, but it's good to know it exists.

Correspondence between StateT and Parser

You may recall what a simple parser type looks like:

```
type Parser a = String -> Maybe (a, String)
```

You may also remember our discussion about the similarities between parsers and `State` in Chapter 24. Now, we could choose to define a `Parser` type in the following manner:

```
newtype StateT s m a =
  StateT { runStateT :: s -> m (a,s) }

type Parser = StateT String Maybe
```

Nobody does this in practice, but it's useful to consider the similarity to get a feel for what `StateT` is all about.

26.6 Types you probably don't want to use

Not every type will necessarily be performant or make sense. `ListT` and `Writer`/`WriterT` are examples of this.

Why not use Writer or WriterT?

It's a bit too easy to get into a situation where `Writer` is either too lazy or too strict for the problem you're trying to solve, and then it'll use more memory than you'd like. `Writer` can accumulate unevaluated thunks, causing memory leaks. It's also inappropriate for logging long-running or ongoing programs due to the fact that you can't retrieve any of the logged values until the computation is complete.[1]

Usually, when `Writer` is used in an application, it's not called `Writer`. Instead, a one-off is created for a specific type `w`. Given that, it's still useful to know when you're looking at something that's a `Reader`, `Writer`, or `State`, even if the author didn't use the types by those names from the `transformers` library. Sometimes, this is because they want a stricter `Writer` than the one already available.

Determining and measuring when more strictness (more eagerly evaluating your thunks) is needed in your programs is the topic of the upcoming chapter on non-strictness.

The ListT you want isn't made from the list type

The most obvious way to implement `ListT` is generally not recommended for a variety of reasons, including:

[1] If you'd like to understand this better, Gabriel Gonzalez has a helpful blog post on the subject: `http://www.haskellforall.com/2014/02/streaming-logging.html`.

1. Most people's first attempt won't pass the associativity law. We're not going to show you a way to write it that does pass that law, because it's not worth it for the reasons listed below.
2. It's not very fast.
3. Streaming libraries like `pipes`[2] and `conduit`[3] do it better for most use cases.

Prior art for "`ListT` done right" also includes `AmbT`[4] by Conal Elliott, although you may find it challenging to understand if you aren't familiar with `ContT` and the motivation behind `Amb`.

Lists in Haskell are as much a control structure as a data structure, so streaming libraries such as `pipes` generally suffice if you need a transformer. This is less of a sticking point in writing applications than you'd think.

26.7 An ordinary type from a transformer

If you have a transformer variant of a type and want to use it as if it were the non-transformer version, you need some `m` structure that doesn't do anything. Have we seen anything like that? What about `Identity`?

```
Prelude> f = runMaybeT $ (+1)
Prelude> f <$> MaybeT (Identity (Just 1))
Identity {runIdentity = Just 2}
Prelude> f <$> MaybeT (Identity Nothing)
Identity {runIdentity = Nothing}
```

Given that we have demonstrated that `Identity` functions as an `m :: * -> *` that does nothing, we can get `Identity` from `IdentityT` and so on in the following manner:

[2] `http://hackage.haskell.org/package/pipes`
[3] `http://hackage.haskell.org/package/conduit`
[4] `https://wiki.haskell.org/Amb`

```
type MyIdentity a = IdentityT Identity a
type Maybe      a = MaybeT Identity a
type Either   e a = EitherT e Identity a
type Reader   r a = ReaderT e Identity a
type State    s a = StateT s Identity a
```

This works fine for recovering the non-transformer variant of each type, as the `Identity` type is acting as a bit of do-nothing structural paste for filling in the gap.

Yeah, but why? You don't ordinarily need to do this if you're working with a transformer that has a corresponding non-transformer type you can use. For example, it's less common to need `ExceptT Identity`, because the `Either` type is already there, so you don't need to retrieve that type from the transformer. However, if you're writing something with, say, `scotty`, where a `ReaderT` is part of the environment, you can't easily retrieve the `Reader` type, because `Reader` is not a type that exists on its own, and you can't modify that `ReaderT` without essentially rewriting all of `scotty`, and, wow, nobody wants that for you. You might then have a situation where what you're doing only needs a `Reader`, not a `ReaderT`, so you could use `ReaderT Identity` to be compatible with `scotty` without having to rewrite everything but still being able to keep your own code a bit tighter and simpler.

The transformers library In general, don't use hand-rolled versions of these transformer types without good reason. You can find many of them in base or the `transformers` library, and that library should have come with your GHC installation.

A note on ExceptT Although a library called `either` exists on Hackage and provides the `EitherT` type, most Haskellers are moving to the identical `ExceptT` type in the `transformers` library. Again, this has mostly to do with the fact that `transformers` comes packaged with GHC already, so `ExceptT` is ready-to-hand; the underlying type is the same.

26.8 Lexically inner is structurally outer

One of the trickier parts of monad transformers is that the lexical representation of the types will violate your intuition with respect to the relationship it has with the structure of your values. Let us note something in the definitions of the following types:

```
-- The definition in transformers may look
-- slightly different. It's not important.
newtype ExceptT e m a =
  ExceptT { runExceptT :: m (Either e a) }

newtype MaybeT m a =
  MaybeT { runMaybeT :: m (Maybe a) }

newtype ReaderT r m a =
  ReaderT { runReaderT :: r -> m a }
```

A necessary byproduct of how transformers work is that the additional structure `m` is always wrapped *around* our value. One thing to note is that it's only wrapped around things we can *have*, not things we *need*, such as with `ReaderT`. The consequence of this is that a series of monad transformers in a type will begin with the innermost type, structurally speaking. Consider the following:

```
module OuterInner where

import Control.Monad.Trans.Except
import Control.Monad.Trans.Maybe
import Control.Monad.Trans.Reader

-- We only need to use return once,
-- because it's one big monad.
embedded :: MaybeT
            (ExceptT String
                     (ReaderT () IO))
            Int
embedded = return 1
```

We can sort of peel away the layers one by one:

```
maybeUnwrap :: ExceptT String
               (ReaderT () IO) (Maybe Int)
maybeUnwrap = runMaybeT embedded

-- Next
eitherUnwrap :: ReaderT () IO
                (Either String (Maybe Int))
eitherUnwrap = runExceptT maybeUnwrap

-- Lastly
readerUnwrap :: ()
             -> IO (Either String
                           (Maybe Int))
readerUnwrap = runReaderT eitherUnwrap
```

Then, if we'd like to evaluate this code, we feed the unit value, `()`, to the function:

```
Prelude> readerUnwrap ()
Right (Just 1)
```

Why is this the result? Consider that we use `return` for a `Monad` comprising `Reader`, `Either`, and `Maybe`:

```
instance Monad ((->) r) where
    return = const

instance Monad (Either e) where
    return = Right

instance Monad Maybe where
    return = Just
```

We can treat having used `return` for the `Reader/Either/Maybe` stack as composition. Consider how we get the same result as `readerUnwrap ()`, here:

```
Prelude> (const . Right . Just $ 1) ()
Right (Just 1)
```

A terminological point to keep in mind when reading about monad transformers is that when Haskellers say *base monad* they usually mean what is structurally outermost:

```
type MyType a = IO [Maybe a]
```

In `MyType`, the base monad is `IO`.

Exercise: Wrap it up

Turn `readerUnwrap` from the previous example back into `embedded` through the use of the data constructors for each transformer. Modify the following code to make it work:

```
embedded :: MaybeT
            (ExceptT String
                     (ReaderT () IO))
            Int
embedded = ??? (const (Right (Just 1)))
```

26.9 MonadTrans

We often want to lift functions into a larger context. We've been doing this for a while with `Functor`, which lifts a function into a context and applies it to the value inside. The facility to do this also undergirds `Applicative`, `Monad`, and `Traversable`. However, `fmap` isn't always enough, so we have some functions that are essentially `fmap` for different contexts:

```
fmap  :: Functor f
      => (a -> b) -> f a -> f b

liftA :: Applicative f
      => (a -> b) -> f a -> f b

liftM :: Monad m
      => (a -> r) -> m a -> m r
```

You might notice the latter two examples have `lift` in the function name. While we've encouraged you not to get too excited about the

meaning of function names, in this case they do give you a clue about what they're doing. They are lifting, just as `fmap` does, a function into some larger context. The underlying structure of the bind function from `Monad` is also a lifting function—`fmap` again!—composed with the crucial `join` function.

In some cases, we want to talk about more or different structure than these types permit. In other cases, we want something that does as much lifting as is necessary to reach some (structurally) outermost position in a stack of monad transformers. Monad transformers can be nested in order to compose various effects into one monster function, but to manage those stacks, we need to lift more.

The type class that lifts

`MonadTrans` is a type class with one, core method: `lift`. Speaking generally, it is about lifting actions in some `Monad` over a transformer type that wraps itself in the original `Monad`. Fancy!

```
class MonadTrans t where
    -- | Lift a computation from
    --   the argument monad to
    --   the constructed monad.
    lift :: (Monad m) => m a -> t m a
```

Here the `t` is a (constructed) monad transformer type that has an instance of `MonadTrans` defined.

We're going to work through a relatively uncomplicated example from `scotty` now.

Motivating MonadTrans

You may remember from previous chapters that `scotty` is a web framework for Haskell. One thing to know about `scotty`, without getting into all the gritty details of how it works, is that the monad transformers the framework relies on are themselves newtypes for monad transformer stacks. Wait, what? Well, look:

```
newtype ScottyT e m a =
  ScottyT
  { runS :: State (ScottyState e m) a }
  deriving (Functor, Applicative, Monad)

newtype ActionT e m a =
  ActionT
  { runAM

    :: ExceptT
         (ActionError e)
         (ReaderT ActionEnv
           (StateT ScottyResponse m))
         a
  }
  deriving ( Functor, Applicative )

type ScottyM = ScottyT Text IO
type ActionM = ActionT Text IO
```

We'll use `ActionM` and `ActionT` and `ScottyM` and `ScottyT` as if they were the same thing, but the `M` variants are type synonyms for the transformers with the inner types already set. This roughly translates to the errors (the left side of the `ExceptT`) in `ScottyM` or `ActionM` being returned as `Text`, while the right side of the `ExceptT`, whatever it does, is `IO`. `ExceptT` is the transformer version of `Either`, and a `ReaderT` and a `StateT` are stacked up inside, as well. These internal mechanics don't matter that much to you, as a user of the `scotty` API, but it's useful to see how much is packed up in there.

Now, back to our example. This is the "hello, world" program using `scotty`, but the following code will cause a type error:

```
-- scotty.hs

{-# LANGUAGE OverloadedStrings #-}

module Scotty where
```

```
import Web.Scotty

import Data.Monoid (mconcat)

main = scotty 3000 $ do
  get "/:word" $ do
    beam <- param "word"
    putStrLn "hello"

    html $
    mconcat ["<h1>Scotty, ",
             beam,
             " me up!</h1>"]
```

Reminder—in your terminal, you can follow along like so:

```
$ stack build scotty
$ stack ghci
Prelude> :l scotty.hs
```

When you try to load it, you should get a type error:

```
• Couldn't match type ‘IO’
    with ‘Web.Scotty.Internal.Types.ActionT
      Data.Text.Internal.Lazy.Text IO’
  Expected type:
    Web.Scotty.Internal.Types.ActionT
      Data.Text.Internal.Lazy.Text IO ()
  Actual type: IO ()
• In a stmt of a 'do' block:
    putStrLn "hello"
  In the second argument of ‘($)’, namely
    ‘do beam <- param "word"
        putStrLn "hello" html (mconcat
          ["<h1>Scotty, ", beam, ....])’
  In a stmt of a 'do' block:
    get "/:word"
      $ do beam <- param "word"
           putStrLn "hello"
           html (mconcat
```

```
            ["<h1>Scotty, ", beam, ....])
  |
9 |         putStrLn "hello"
  |         ^^^^^^^^^^^^^^^^
```

The reason for this type error is that `putStrLn` has the type `IO ()`, but it is inside a `do` block inside our `get`, and the monad that code is in is therefore `ActionM`/`ActionT`:

```
get :: RoutePattern
    -> ActionM ()
    -> ScottyM ()
```

Our `ActionT` type eventually reaches `IO`, but there's additional structure we need to lift over first. To fix this, we'll start by adding an import:

```
import Control.Monad.Trans.Class
```

And amend that line with `putStrLn` to the following:

```
lift (putStrLn "hello")
```

It should work.

You can assert a type for the `lift` embedded in the `scotty` action:

```
let hello = putStrLn "hello"
(lift :: IO a -> ActionM a) hello
```

Let's see what it does. Load the file again and call the `main` function. You should see this message:

```
Setting phasers to stun...
  (port 3000) (ctrl-c to quit)
```

In the address bar of your web browser, type `localhost:3000`. You should notice two things: one is that there is nothing in the `beam` slot of the text that prints to your screen, and the other is that it prints `"hello"` to your terminal where the program is running. Try adding a word to the end of the address:

```
localhost:3000/beam
```

The text on your screen should change, and `"hello"` should print in your terminal again. That `/:word` parameter is what has been bound via the variable `beam` into that HTML line at the end of the `do` block, while the `"hello"` has been lifted over the `ActionM` so that it can print in your terminal. It will print another `"hello"` to your terminal every time something happens on the web page.

We can concretize our use of `lift` in the following steps. Please follow along by asserting the types for the application of `lift` in the `scotty` application above:

```
lift :: (Monad m, MonadTrans t)
     => m a -> t m a
lift :: (MonadTrans t)
     => IO a -> t IO a
lift :: IO a -> ActionM a
lift :: IO () -> ActionM ()
```

We go from `(t IO a)` to `(ActionM a)`, because the `IO` is inside the `ActionM`. Let's examine `ActionM` more carefully:

```
Prelude> import Web.Scotty
Prelude> import Web.Scotty.Trans
Prelude> :info ActionM
type ActionM = ActionT Data.Text.Internal.Lazy.Text IO
    -- Defined in Web.Scotty
```

We can see what this `lift` does by looking at the `MonadTrans` instance for `ActionT`, which is what `ActionM` is a type alias of:

```
instance MonadTrans (ActionT e) where
    lift = ActionT . lift . lift . lift
```

Part of the niceness here is that `ActionT` is itself defined in terms of *three more* monad transformers. We can see this in the definition of `ActionT`:

```
newtype ActionT e m a =
ActionT {
  runAM
    :: ExceptT
         (ActionError e)
         (ReaderT ActionEnv
           (StateT ScottyResponse m))
         a
  } deriving (Functor, Applicative)
```

Let's first replace the `lift` for `ActionT` with its definition and see if it still works:

```
{-# LANGUAGE OverloadedStrings #-}

module Scotty where

import Web.Scotty
import Web.Scotty.Internal.Types
  (ActionT(..))
import Control.Monad.Trans.Class
import Data.Monoid (mconcat)
```

All that the `(..)` in the code above means is that we want to import all the data constructors of the `ActionT` type, rather than none or a particular list of them. You can look into the syntax in more detail independently, if you like. Now for the `scotty` application itself:

```
main = scotty 3000 $ do
  get "/:word" $ do
    beam <- param "word"

    (ActionT . lift . lift . lift)
      (putStrLn "hello")
    html $
      mconcat ["<h1>Scotty, ",
               beam,
               " me up!</h1>"]
```

This should still work! Note that we had to ask for the data constructor for `ActionT` from an `Internal` module, because the implementation is hidden by default. We've got three lifts, one each for `ExceptT`, `ReaderT`, and `StateT`.

Next, we'll do `ExceptT`:

```
instance MonadTrans (ExceptT e) where
    lift = ExceptT . liftM Right
```

To use that in our code, add the following import:

```
import Control.Monad.Trans.Except
```

And our app changes into the following:

```
main = scotty 3000 $ do
  get "/:word" $ do
    beam <- param "word"

    (ActionT
     . (ExceptT . liftM Right)
     . lift
     . lift) (putStrLn "hello")
    html $
      mconcat ["<h1>Scotty, ",
               beam,
               " me up!</h1>"]
```

We take a gander at the `Reader` module in the `transformers` library and see the following:

```
instance MonadTrans (ReaderT r) where
  lift = liftReaderT

liftReaderT :: m a -> ReaderT r m a
liftReaderT m = ReaderT (const m)
```

For unknown reasons, `liftReaderT` isn't exported by `transformers`, but we can redefine it ourselves. Add the following to the module:

```
import Control.Monad.Trans.Reader
```

```
liftReaderT :: m a -> ReaderT r m a
liftReaderT m = ReaderT (const m)
```

Then, our app can be defined as follows:

```
main = scotty 3000 $ do
  get "/:word" $ do
    beam <- param "word"
    (ActionT
     . (ExceptT . fmap Right)
     . liftReaderT
     . lift
     ) (putStrLn "hello")

    html $
      mconcat ["<h1>Scotty, ",
               beam,
               " me up!</h1>"]
```

Or, instead of `liftReaderT`, we could have done this:

```
     . (\m -> ReaderT (const m))
```

Or:

```
    (ActionT
     . (ExceptT . fmap Right)
     . ReaderT . const
     . lift
     ) (putStrLn "hello")
```

Now for that last `lift` over `StateT`! Remembering that it was the *lazy* `StateT` that the type of `ActionT` mentions, we see the following `MonadTrans` instance:

```
instance MonadTrans (StateT s) where
    lift m = StateT $ \s -> do
        a <- m
        return (a, s)
```

First, let's get our import in place:

```
import Control.Monad.Trans.State.Lazy
  hiding (get)
```

We need to hide `get`, because `scotty` already has a different `get` function defined and we don't need the one from `StateT`. Then, inlining that into our app code:

```
main = scotty 3000 $ do
  get "/:word" $ do
    beam <- param "word"

    (ActionT
     . (ExceptT . fmap Right)
     . ReaderT . const
     . \m -> StateT (\s -> do
                      a <- m
                      return (a, s))
      ) (putStrLn "hello")

    html $
      mconcat ["<h1>Scotty, ",
               beam,
               " me up!</h1>"]
```

Note that we need an outer lambda before the `StateT` in order to get the monadic action we are lifting. At this point, we're in the outermost position we can be, and since `ActionM` defines `ActionT`'s outermost monadic type as being `IO`, that means our `putStrLn` works fine after all this lifting.

Typically, a `MonadTrans` instance lifts over only one layer at a time, but `scotty` abstracts away the underlying structure so that you don't have to care. That's why it goes ahead and does the next three lifts for you. The critical thing to realize here is that lifting means you're embedding an expression in a larger context by adding structure that doesn't do anything.

MonadTrans instances

Now you see why we have `MonadTrans` and have a picture of what `lift`, the only method of `MonadTrans`, does.

Here are some examples of `MonadTrans` instances:

1. `IdentityT`

```
instance MonadTrans IdentityT where
    lift = IdentityT
```

2. `MaybeT`

```
instance MonadTrans MaybeT where
    lift = MaybeT . liftM Just
```

```
lift
  :: (Monad m)
  => m a -> t m a
(MaybeT . liftM Just)
  :: Monad m
  => m a -> MaybeT m a

MaybeT
  :: m (Maybe a) -> MaybeT m a
(liftM Just)
  :: Monad m
  => m a -> m (Maybe a)
```

Roughly speaking, this has taken an `m a` and lifted it into a `MaybeT` context.

The general pattern with `MonadTrans` instances demonstrated by `MaybeT` is that you're usually going to lift the injection of the known structure (with `MaybeT`, the known structure is `Maybe`) over some `Monad`. Injection of structure usually means `return`, but since with `MaybeT` we know we want a `Maybe` structure, we use `Just`. That transforms an `m a` into `m (T a)`, where capital `T` is some concrete type you're lifting the `m a` into. Then, to cap it all off, you use the data constructor for your monad transformer, and the value is now lifted into the larger context. Here's a summary of the stages the type of the value goes through:

```
v :: Monad m => m a
liftM Just :: Monad m => m a -> m (Maybe a)
liftM Just v :: m (Maybe a)
MaybeT (liftM Just v) :: MaybeT m a
```

See if you can work out the types of this one:

3. `ReaderT`

```
instance MonadTrans (ReaderT r) where
  lift = ReaderT . const
```

And now, write some instances!

Exercises: Lift more

Keep in mind what these are doing, follow the types, lift till you drop.

1. You thought you were done with `EitherT`:

```
instance MonadTrans (EitherT e) where
    lift = undefined
```

2. Or `StateT`. This one'll be more obnoxious. It's fine if you've seen this before:

```
instance MonadTrans (StateT s) where
    lift = undefined
```

Prolific lifting is the failure mode

Apologies to the original authors, but sometimes with the use of concretely and explicitly typed monad transformers, you'll see stuff like this:

```
addSubWidget :: (YesodSubRoute sub master)
               => sub
               -> WidgetT sub master a
               -> WidgetT sub' master a
addSubWidget sub w =
```

```
do master <- liftHandler getYesod
   let sr = fromSubRoute sub master
   i <- WidgetT $ lift $ lift $ lift
                $ lift $ lift $ lift
                $ lift get

   w' <- liftHandler
         $ toMasterHandlerMaybe sr
           (const sub) Nothing
         $ flip runStateT i $ runWriterT
         $ runWriterT $ runWriterT

         $ runWriterT $ runWriterT
         $ runWriterT $ runWriterT
         $ unWidgetT w

   let (((((((a,
               body),
              title),
             scripts),
            stylesheets),
           style),
          jscript),
         h),
        i') = w'

   WidgetT $ do
     tell body
     lift $ tell title
     lift $ lift $ tell scripts
     lift $ lift $ lift
       $ tell stylesheets
     lift $ lift $ lift $ lift
       $ tell style
```

```
      lift $ lift $ lift $ lift $ lift
        $ tell jscript
      lift $ lift $ lift $ lift $ lift
        $ lift $ tell h
      lift $ lift $ lift $ lift
           $ lift $ lift $ lift $ put i'
      return a
```

Do *not* write code like this. More to the point, do not write code like this and then proceed to blog about how terrible monad transformers are.

Wrap it, smack it, pre-lift it

OK, so how do we avoid that horror show? Well, there are a lot of ways, but one of the most robust and common is newtyping your `Monad` stack and abstracting away the representation. From there, you provide the functionality leveraging the representation as part of your API. A good example of this comes to us from… `scotty`!

Let's take a gander at the `ActionM` type we mentioned earlier. Again, to make the type read more nicely, we import some other modules, as well:

```
Prelude> import Web.Scotty
Prelude> import Data.Text.Lazy
Prelude> :info ActionM
type ActionM =
  Web.Scotty.Internal.Types.ActionT Text IO
    -- Defined in Web.Scotty
```

`scotty` hides the underlying type by default, because you ordinarily wouldn't care or think about it in the course of writing your application. What `scotty` does here is good practice. This design keeps the underlying implementation hidden by default but lets us import an `Internal` module to get at the representation in case we need to (with more modules to clean up the types):

```
Prelude> import Web.Scotty.Internal.Types
Prelude> import Control.Monad.Trans.Reader
Prelude> import Control.Monad.Trans.State.Lazy
```

```
Prelude> import Control.Monad.Trans.Except

Prelude> :info ActionT
type role ActionT
  nominal representational nominal
newtype ActionT e (m :: * -> *) a
  = ActionT
      {runAM :: ExceptT
                (ActionError e)
                (ReaderT ActionEnv
                  (StateT ScottyResponse m))
                a}
instance (Monad m, ScottyError e)
      => Monad (ActionT e m)
instance Functor m => Functor (ActionT e m)
instance Monad m => Applicative (ActionT e m)
```

What's nice about this approach is that it subjects the consumers (which could include yourself) of your type to less noise within an application. It also doesn't require reading papers written by people trying *very* hard to impress a thesis advisor, although poking through prior art for ideas is recommended. It can reduce or eliminate manual lifting within the `Monad`, as well. Note that we only have to use `lift` once to perform an IO action in `ActionM`, even though the underlying implementation has more than one transformer flying around.

26.10 MonadIO, aka zoom-zoom

There's more than one way to skin a cat, and there's more than one way to lift an action over additional structure. `MonadIO` is a different design than `MonadTrans`, because rather than lifting through one layer at a time, `MonadIO` is intended to keep lifting your IO action until it is lifted over *all* structure embedded in the outermost `IO` type. The idea here is that you'd write `liftIO` *once*, and it would instantiate to all of the following types:

```
liftIO :: IO a -> ExceptT e IO a
liftIO :: IO a -> ReaderT r IO a
liftIO :: IO a -> StateT s IO a
```

```
-- As Sir Mix-A-Lot once said,
-- stack 'em up deep.
liftIO :: IO a -> StateT s (ReaderT r IO) a
liftIO :: IO a
       -> ExceptT
            e
            (StateT s (ReaderT r IO))
            a
```

You don't have to lift multiple times if you're trying to reach a base (outermost) `Monad` that happens to be `IO`, because you have `liftIO`.

In the `transformers` library, the `MonadIO` class resides in the module `Control.Monad.IO.Class`:

```
class (Monad m) => MonadIO m where
    -- | Lift a computation
    --   from the 'IO' monad.
    liftIO :: IO a -> m a
```

The commentary within the module is reasonably helpful, though it doesn't highlight what makes `MonadIO` different from `MonadTrans`:

> Monads in which `IO` computations may be embedded. Any monad built by applying a sequence of monad transformers to the `IO` monad will be an instance of this class.
>
> Instances should satisfy the following laws, which state that `liftIO` is a transformer of monads:
>
> 1. `liftIO . return = return`
> 2. `liftIO (m >>= f) =`
>
>   `liftIO m >>= (liftIO . f)`

Let us modify the `scotty` example app to print a string:

```
{-# LANGUAGE OverloadedStrings #-}

module Main where

import Web.Scotty
```

```
import Control.Monad.IO.Class
import Data.Monoid (mconcat)

main = scotty 3000 $ do
  get "/:word" $ do
    beam <- param "word"
    liftIO (putStrLn "hello")

    html $
      mconcat ["<h1>Scotty, ",
               beam,
               " me up!</h1>"]
```

If you then run `main` in a REPL or build a binary and execute it, you'll be able to request a response from the server using your web browser (as we showed you earlier) or a command line application like `curl`. If you use a browser and see `"hello"` printed more than once, it's likely your browser made the request more than once. You shouldn't see this behavior if you test it with `curl`.

Example MonadIO instances

1. `IdentityT`

```
instance (MonadIO m)
      => MonadIO (IdentityT m) where
    liftIO = IdentityT . liftIO
```

2. `EitherT`

```
instance (MonadIO m)
      => MonadIO (EitherT e m) where
    liftIO = lift . liftIO
```

Exercises: Some instances

1. `MaybeT`

```
instance (MonadIO m)
      => MonadIO (MaybeT m) where
    liftIO = undefined
```

2. `ReaderT`

```
instance (MonadIO m)
      => MonadIO (ReaderT r m) where
    liftIO = undefined
```

3. `StateT`

```
instance (MonadIO m)
      => MonadIO (StateT s m) where
    liftIO = undefined
```

Hint: your instances should be simple.

26.11 Monad transformers in use

MaybeT in use

These are some example of `MaybeT` in use; we will not comment on them and instead let you research them further yourself, if you want. The origins of the code are noted in the samples:

```
-- github.com/wavewave/hoodle-core
recentFolderHook
  :: MainCoroutine (Maybe FilePath)

recentFolderHook = do
  xstate <- get

  (r :: Maybe FilePath) <- runMaybeT $ do
    hset <- hoist (view hookSet xstate)

    rfolder <-
      hoist (H.recentFolderHook hset)
    liftIO rfolder
  return r

-- github.com/devalot/hs-exceptions
-- src/maybe.hs
```

```
addT :: FilePath
     -> FilePath
     -> IO (Maybe Integer)

addT f1 f2 = runMaybeT $ do
  s1 <- sizeT f1
  s2 <- sizeT f2
  return (s1 + s2)

-- wavewave/ghcjs-dom-delegator
-- example/Example.hs
main :: IO ()
main = do

  clickbarref <-
    asyncCallback1 AlwaysRetain clickbar
  clickbazref <-
    asyncCallback1 AlwaysRetain clickbaz

  r <- runMaybeT $ do
    doc <- MaybeT currentDocument
    bar <- lift . toJSRef
           =<< MaybeT
                 (documentQuerySelector doc
                   (".bar" :: JSString))

    baz <- lift . toJSRef
           =<< MaybeT
                 (documentQuerySelector doc
                   (".baz" :: JSString))

    lift $ do
      ref <- newObj
      del <-delegator ref
      addEvent bar "click" clickbarref
      addEvent baz "click" clickbazref

  case r of
    Nothing -> print "something wrong"
    Just _ -> print "welldone"
```

Temporary extension of structure

Although we commonly think of monad transformers as being used to define one big context for an application, particularly with things like `ReaderT`, there are other ways. One pattern that is often useful is temporarily extending additional structure to avoid boilerplate. Here's an example using plain old `Maybe` and `scotty`:

```
{-# LANGUAGE OverloadedStrings #-}

module Main where

import Control.Monad.IO.Class
import Data.Maybe (fromMaybe)
import Data.Text.Lazy (Text)
import Web.Scotty

param' :: Parsable a
       => Text -> ActionM (Maybe a)
param' k = rescue (Just <$> param k)
                  (const (return Nothing))

main = scotty 3000 $ do
  get "/:word" $ do

    beam' <- param' "word"
    let beam = fromMaybe "" beam'

    i <- param' "num"
    liftIO $ print (i :: Maybe Integer)

    html $
    mconcat ["<h1>Scotty, ",
             beam,
             " me up!</h1>"]
```

This works well enough but could get tedious in a hurry if we had a bunch of stuff that returns `ActionM (Maybe ...)` and we want to short-circuit the moment any of them fail. So, we do something similar but with `MaybeT`, building up more data in one go:

```
{-# LANGUAGE OverloadedStrings #-}

module Main where

import Control.Monad.IO.Class
import Control.Monad.Trans.Class
import Control.Monad.Trans.Maybe
import Data.Maybe (fromMaybe)
import Data.Text.Lazy (Text)
import Web.Scotty

param' :: Parsable a
       => Text -> MaybeT ActionM a
param' k = MaybeT $
           rescue (Just <$> param k)
                  (const (return Nothing))

type Reco =
  (Integer, Integer, Integer, Integer)

main = scotty 3000 $ do
  get "/:word" $ do
    beam' <- param "word"

    let beam = fromMaybe "" beam'
    reco <- runMaybeT $ do
      a <- param' "1"
      liftIO $ print a

      b <- param' "2"
      c <- param' "3"
      d <- param' "4"
      (lift . lift) $ print b
      return ((a, b, c, d) :: Reco)

    liftIO $ print reco
    html $
    mconcat ["<h1>Scotty, ",
             beam,
             " me up!</h1>"]
```

Some important things to note here:

1. We only have to use `liftIO` once, even in the presence of additional structure, whereas with `lift` we have to lift twice to address `MaybeT` and `ActionM`.
2. The *one big bind* of the `MaybeT` means we could take the existence of `a`, `b`, `c`, and `d` for granted in that context, but the `reco` value itself is `Maybe Reco`, because any part of the computation could fail in the absence of the needed parameter.
3. It knows which monad we mean for that `do` block because of the `runMaybeT` in front of the `do`. This serves the dual purpose of unpacking the `MaybeT` into an `ActionM (Maybe Reco)`, which we can bind out into `Maybe Reco`.

ExceptT, aka EitherT, in use

The example with `Maybe` and `scotty` may not have totally satisfied, because the failure mode isn't helpful to an end user—all they know is `Nothing`. Accordingly, `Maybe` is usually something that should get handled early and often in a place local to where it is produced, so that you avoid mysterious `Nothing` values floating around and short-circuiting your code. They're not something you want to return to end users, either. Fortunately, we have `Either` for more descriptive short-circuiting computations!

Scotty, again

We'll use `scotty` again to demonstrate this. Once again, we'll show you a plain example:

```
{-# LANGUAGE OverloadedStrings #-}

module Main where

import Control.Monad.IO.Class
import Data.Text.Lazy (Text)
import Web.Scotty
```

```
param' :: Parsable a
       => Text -> ActionM (Either String a)
param' k =

  rescue (Right <$> param k)
         (const
          (return

           (Left $ "The key: "
                   ++ show k
                   ++ " was missing!")))

main = scotty 3000 $ do
  get "/:word" $ do
    beam <- param "word"
    a <- param' "1"
    let a' = either (const 0) id a

    liftIO $ print (a :: Either String Int)
    liftIO $ print (a' :: Int)
    html $
    mconcat ["<h1>Scotty, ",
             beam,
             " me up!</h1>"]
```

Note that we have to manually fold the `Either` if we want to address the desired `Int` value. Try to avoid having default fallback values in real code, though. This could get nutty in a hurry if we have many things we're pulling out of the parameters.

Let's do that but with `ExceptT` from `transformers`. Remember, `ExceptT` is another name for `EitherT`:

```
{-# LANGUAGE OverloadedStrings #-}

module Main where
```

```
import Control.Monad.IO.Class
import Control.Monad.Trans.Class
import Control.Monad.Trans.Except
import Data.Text.Lazy (Text)
import qualified Data.Text.Lazy as TL
import Web.Scotty

param' :: Parsable a
       => Text -> ExceptT String ActionM a
param' k =

  ExceptT $
  rescue (Right <$> param k)
         (const
          (return
           (Left $ "The key: "
                   ++ show k
                   ++ " was missing!")))

type Reco =
  (Integer, Integer, Integer, Integer)

tshow = TL.pack . show

main = scotty 3000 $ do
  get "/" $ do
    reco <- runExceptT $ do

      a <- param' "1"
      liftIO $ print a
      b <- param' "2"
      c <- param' "3"
      d <- param' "4"
      (lift . lift) $ print b
      return ((a, b, c, d) :: Reco)

    case reco of
      (Left e) -> text (TL.pack e)
      (Right r) ->
```

```
        html $
        mconcat ["<h1>Success! Reco was: ",
                 tshow r,
                 "</h1>"]
```

If you pass it a request like this:

```
http://localhost:3000/?1=1
```

It'll ask for the parameter `"2"`, because that is the next param you ask for after `"1"`.

If you pass it a request like this:

```
http://localhost:3000/?1=1&2=2&3=3&4=4
```

You should see this response in your browser or terminal:

```
Success! Reco was: (1,2,3,4)
```

As before, we get to benefit from *one big bind* under the `ExceptT`.

Slightly more advanced code

From some code by Sean Chalmers.[5] Some context for the `EitherT` application you'll see:

```
type Et a = EitherT SDLErr IO a

mkWindow :: HasSDLErr m =>
            String
         -> CInt -> CInt
         -> m SDL.Window
mkRenderer :: HasSDLErr m
           => SDL.Window -> m SDL.Renderer

hasSDLErr :: (MonadIO m, MonadError e m)
          => (a -> b)
          -> (a -> Bool)
          -> e -> IO a -> m b
```

[5] https://github.com/mankyKitty/Meteor/

```
hasSDLErr g f e a =
  liftIO a
  >>= \r ->
        bool (return $ g r)
             (throwError e) $ f r

class (MonadIO m, MonadError SDLErr m)
   => HasSDLErr m where
  decide  :: (a -> Bool)
          -> SDLErr -> IO a -> m a
  decide' :: (Eq n, Num n)
          => SDLErr -> IO n -> m ()

instance HasSDLErr
  (EitherT SDLErr IO) where
    decide  = hasSDLErr id
    decide' = hasSDLErr (const ()) (/= 0)
```

Then, in use:

```
initialise :: Et (SDL.Window,SDL.Renderer)
initialise = do
  initSDL [SDL.SDL_INIT_VIDEO]
  win <-
    mkWindow "Meteor!"
             screenHeight
             screenWidth
  rdr <- mkRenderer win
  return (win,rdr)

createMeteor :: IO (Either SDLErr MeteorS)
createMeteor = do
  eM <- runEitherT initialise
  return $ mkMeteor <$> eM
```

```
  where
    emptyBullets = V.empty

    mkMeteor (w,r) = MeteorS w r
                     getInitialPlayer
                     emptyBullets
                     getInitialMobs
                     False
```

26.12 Monads do not commute

Remember that monads in general do not commute, and you aren't guaranteed something sensible for every possible combination of types. The kit we have for constructing and using monad transformers is useful but is not a license to *not think*!

Hypothetical exercise

Consider `ReaderT r Maybe` and `MaybeT (Reader r)`—are these types equivalent? Do they do the same thing? Try writing otherwise similar bits of code with each, and see if you can prove they're the same or different.

26.13 Transform if you want to

If you find monad transformers difficult or annoying, then don't bother! Most of the time you can get by with `liftIO` and plain IO actions, functions, `Maybe` values, etc. Do the simplest (for you) thing first when mapping out something new or unfamiliar. It's better to let more structured formulations of programs fall out naturally from having kicked around something uncomplicated than to blow out your working memory budget in one go. Don't worry about seeming unsophisticated; in our opinion, being happy and productive is better than being fancy.

Keep it basic in your first attempt. Never make it more elaborate initially than is strictly necessary. You'll figure out when the transformer variant of a type will save you complexity in the process of writing your programs. We have taken you through these topics,

because you'll need at least a passing familiarity with them in order to use modern Haskell libraries or frameworks, but it's not a design dictate you must follow.

26.14 Chapter exercises

Write the code

1. `rDec` is a function that should get its argument in the context of `Reader` and return a value decremented by one:

```
rDec :: Num a => Reader a a
rDec = undefined
```

```
Prelude> import Control.Monad.Trans.Reader
Prelude> runReader rDec 1
0
Prelude> fmap (runReader rDec) [1..10]
[0,1,2,3,4,5,6,7,8,9]
```

   Note that `Reader` from `transformers` is the `ReaderT` of `Identity` and that `runReader` is a convenience function that throws away the meaningless structure for you. Play with `runReaderT` if you like.

2. Once you have an `rDec` that works, make it and any inner lambdas point-free, if that's not already the case.

3. `rShow` is `show`, but in `Reader`:

```
rShow :: Show a
      => ReaderT a Identity String
rShow = undefined
```

```
Prelude> runReaderT rShow 1
"1"
Prelude> fmap (runReaderT rShow) [1..10]
["1","2","3","4","5","6","7","8","9","10"]
```

4. Once you have an `rShow` that works, make it point-free.

5. `rPrintAndInc` will first print the input with a greeting, then return the input incremented by one:

```
rPrintAndInc :: (Num a, Show a)
             => ReaderT a IO a
rPrintAndInc = undefined

Prelude> runReaderT rPrintAndInc 1
Hi: 1
2
Prelude> traverse (runReaderT rPrintAndInc) [1..10]
Hi: 1
Hi: 2
Hi: 3
Hi: 4
Hi: 5
Hi: 6
Hi: 7
Hi: 8
Hi: 9
Hi: 10
[2,3,4,5,6,7,8,9,10,11]
```

6. `sPrintIncAccum` first prints the input with a greeting, then "puts" the incremented input as the new state and returns the original input as a `String`:

```
sPrintIncAccum :: (Num a, Show a)
               => StateT a IO String
sPrintIncAccum = undefined

Prelude> runStateT sPrintIncAccum 10
Hi: 10
("10",11)
Prelude> mapM (runStateT sPrintIncAccum) [1..5]
Hi: 1
Hi: 2
Hi: 3
Hi: 4
```

```
Hi: 5
[("1",2),("2",3),("3",4),("4",5),("5",6)]
```

Fix the code

The code won't type check as written; fix it so that it does. Feel free to add imports if they provide something useful. Functions are used that we haven't introduced. You're not allowed to change the types asserted. You may have to fix the code in more than one place:

```
import Control.Monad.Trans.Maybe
import Control.Monad

isValid :: String -> Bool
isValid v = '!' `elem` v

maybeExcite :: MaybeT IO String
maybeExcite = do
    v <- getLine
    guard $ isValid v
    return v

doExcite :: IO ()
doExcite = do
  putStrLn "say something excite!"
  excite <- maybeExcite

  case excite of
    Nothing -> putStrLn "MOAR EXCITE"
    Just e ->
      putStrLn
        ("Good, was very excite: " ++ e)
```

Hit counter

We're going to provide an initial scaffold of a `scotty` application that counts hits to specific URIs. It also prefixes the keys with a prefix defined on app initialization, retrieved via command line arguments:

```
{-# LANGUAGE OverloadedStrings #-}

module Main where

import Control.Monad.Trans.Class
import Control.Monad.Trans.Reader
import Data.IORef
import qualified Data.Map as M

import Data.Maybe (fromMaybe)
import Data.Text.Lazy (Text)
import qualified Data.Text.Lazy as TL
import System.Environment (getArgs)
import Web.Scotty.Trans

data Config =
  Config {
    -- that's one, one click!
    -- two... two clicks!
    -- Three BEAUTIFUL clicks! ah ah ahhhh
    counts :: IORef (M.Map Text Integer)
  , prefix :: Text
  }
```

Stuff inside `ScottyT` is, except for things that escape via `IO`, effectively read-only, so we can't use `StateT`. It would overcomplicate things to attempt to do so, and you should be using a proper database for production applications:

```
type Scotty =
  ScottyT Text (ReaderT Config IO)
type Handler =
  ActionT Text (ReaderT Config IO)

bumpBoomp :: Text
          -> M.Map Text Integer
          -> (M.Map Text Integer, Integer)
bumpBoomp k m = undefined
```

```
app :: Scotty ()
app =
  get "/:key" $ do
    unprefixed <- param "key"
    let key' = mappend undefined unprefixed
    newInteger <- undefined

    html $
    mconcat [ "<h1>Success! Count was: "
            , TL.pack $ show newInteger
            , "</h1>"
            ]

main :: IO ()
main = do
  [prefixArg] <- getArgs
  counter <- newIORef M.empty
  let config = undefined
      runR = undefined
  scottyT 3000 runR app
```

Code is missing and broken. Your task is to make it work. Do whatever is necessary. You should be able to run the server from inside of GHCi, passing arguments like so:

```
Prelude> :main lol
Setting phasers to stun... (port 3000) (ctrl-c to quit)
```

You could also build a binary and pass the arguments from your shell, but do what you like. Once it's running, you should be able to bump the counts like so:

```
$ curl localhost:3000/woot
<h1>Success! Count was: 1</h1>
$ curl localhost:3000/woot
<h1>Success! Count was: 2</h1>
$ curl localhost:3000/blah
<h1>Success! Count was: 1</h1>
```

Note that the underlying "key" used in the counter when you `GET /woot` is `"lolwoot"`, because we pass `"lol"` to `main`. For a giggle, try the URI for one of the keys in your browser, and mash refresh a bunch.

If you get stuck, consider checking for examples, such as the `reader` file in the `examples` directory in the `scotty` repository on GitHub.[6]

Morra

1. Write the game Morra[7] using `StateT` and `IO`. The state being accumulated is the score of the player and the computer AI opponent. To start, make the computer choose its play randomly. On exit, report the scores for the player and the computer, congratulating the winner.

2. Add a human vs. human mode to the game with interstitial screens between input prompts so the players can change out of the hot seat without seeing the other player's answer.

3. Improve the computer AI slightly by making it remember 3-grams of the player's behavior, adjusting its answer instead of deciding randomly when the player's behavior matches a known behavior. For example:

```
-- P is Player
-- C is Computer
-- Player is odds, Computer is evens.
P: 1
C: 1
- C wins
P: 2
C: 1
- P wins
P: 2
C: 1
- P wins
```

[6] `https://github.com/scotty-web/scotty/tree/master/examples`

[7] You can find descriptions of the rules and gameplay of Morra online.

At this point, the computer should register the pattern (1, 2, 2)—in other words, the player picked 2 after both 1 and 2. The next time the player picks 1 followed by 2, the computer should assume the next play will be 2 and pick 2, in order to win.

4. The 3-gram thing is pretty simple and dumb. Humans are bad at being random; they often have sub-patterns in their moves.

26.15 Definition

In general, the term *leak* refers to something that consumes a resource in a way that renders it unusable or irrecoverable. Specifically, when we talk about a *memory leak*, we're talking about consuming memory in a way that renders it unusable or unrecoverable by other programs or parts of a program. This can happen if your program is written in such a way that it accumulates large amounts of unevaluated thunks or holds in memory a reference to something that it's not using anymore. The garbage collector cannot sweep those things away, so the amount of memory a program is using can increase, sometimes rapidly and alarmingly, while the amount of available or free memory decreases.

26.16 Follow-up resources

1. Simon Marlow. *Parallel and Concurrent Programming in Haskell*. `https://simonmar.github.io/pages/pcph.html`

Chapter 27

Non-strictness

> Progress doesn't come from early risers—progress is made by lazy men looking for easier ways to do things.
>
> Robert A. Heinlein

27.1 Laziness

This chapter concerns the ways in which Haskell programs are evaluated. We addressed this a bit in previous chapters, for example, in Chapter 10, on folding lists, where we went into some detail about how folds evaluate. In this chapter, our goal is to give you enough information about Haskell's evaluation strategy that you'll be able to reason confidently about the reduction process of your expressions and introduce stricter evaluation where you want it.

Most programming languages have strict evaluation semantics. Haskell technically has *non-strict*—not lazy—evaluation, but the difference between lazy and non-strict is not practically relevant, so you'll hear Haskell referred to as either a lazy language or a non-strict one.

A very rough outline of Haskell's evaluation strategy is this: most expressions are only reduced, or evaluated, when necessary. When the evaluation process begins, a *thunk* is created for each expression. We'll go into more detail about this shortly, but a thunk is like a placeholder in the underlying graph of a program. Whatever expression the thunk is holding a place for can be evaluated when necessary, but if it's never needed, it never gets reduced, and then the garbage collector comes along and sweeps it away. If it is evaluated, because it's in a graph, it can often be shared between expressions—that is, once `x = 1 + 1` has been evaluated, anytime `x` is forced, it does not have to be re-computed.

This is the laziness of Haskell: don't do more work than needed. Don't evaluate until necessary. Don't re-evaluate if you don't have to. We'll go through the details of how this works, exceptions to the general principles, and how to control evaluation by adding strictness where desired.

Specifically, we will:

- Define call-by-name and call-by-need evaluation.
- Explain the main effects of non-strict evaluation.
- Live the Thunk Life.[1]

[1]We love you, Jesse!

- Consider the runtime behavior of non-strict code in terms of sharing.
- Develop methods for observing sharing behavior and measuring program efficiency.
- Bottom out with the bottoms.

27.2 Observational bottom theory

In our discussion of non-strictness in Haskell, we're going to have a great deal to say about bottom.[2] This is partly because non-strictness is *defined* by the ability to evaluate expressions that have bottom in them, as long as the bottom itself is never forced. Bottoms also give us a convenient method of observing evaluation in Haskell. By causing the program to halt immediately with an error, bottom serves as our first means of understanding non-strictness. You probably recall that we have used this trick before.

Standards and obligations

Technically, Haskell is only obligated to be non-strict, not lazy. A truly lazy language *memoizes*, or holds in memory, the results of all the functions it does evaluate, and, outside of toy programs, this tends to use unacceptably large amounts of memory. Implementations of Haskell, such as GHC Haskell, are only obligated to be non-strict, such that they have the same behavior with respect to bottom; they are not required to take a particular approach to how the program executes or how efficiently it does so.

The essence of non-strictness is that you can have an expression that evaluates to a value, even if bottom or infinite data lurks within. For example, the following would only work in a non-strict language:

```
Prelude> fst (1, undefined)
1
Prelude> snd (undefined, 2)
2
```

[2] Observational bottom theory is not a real thing. Do not email us about this.

The idea is that any given implementation of non-strictness is acceptable as long as it respects when it's supposed to return a value successfully or bottom out.

27.3 Outside in, inside out

Strict languages evaluate *inside out*; non-strict languages like Haskell evaluate *outside in*. Outside in means that evaluation proceeds from the outermost parts of expressions and works inward based on what values are forced. This means the order of evaluation and what gets evaluated can vary depending on inputs.

The following example is written in a slightly arcane way to make the evaluation order more obvious:

```
possiblyKaboom =
  \f -> f fst snd (0, undefined)

-- Booleans in lambda form
true :: a -> a -> a
true  = \a -> (\b -> a)

false :: a -> a -> a
false = \a -> (\b -> b)
```

When we apply `possiblyKaboom` to `true`, `true` is the `f`, `fst` is the `a`, and `snd` is the `b`. Semantically, case matches, guards expression, and if-then-else expressions could all be rewritten in this manner (they are not in fact decomposed this way by the compiler), by nesting lambdas and reducing them from the outside in:

```
(\f ->
  f fst snd (0, undefined))
  (\a -> (\b -> a))
(\a -> (\b -> a)) fst snd (0, undefined)
(\b -> fst) snd (0, undefined)
fst (0, undefined)
0
```

The next example is written in more normal Haskell but will return the same result. When we apply the function to `True` here, we case on the `True` to return the first value of the tuple:

```
possiblyKaboom b =
  case b of
    True ->  fst tup
    False -> snd tup
  where tup = (0, undefined)
```

The bottom is inside a tuple, and the tuple is bound inside of a lambda that cases on a Boolean value and returns either the first or second element of the tuple. Since we start evaluating from the outside, as long as this function is only ever applied to `True`, that bottom will never cause a problem. However, at the risk of stating the obvious, we do not encourage you to write programs with bottoms lying around willy-nilly.

When we say evaluation works outside in, we're talking about evaluating a series of nested expressions, and not only are we starting from the outside and working in, but we're also only evaluating some of the expressions some of the time. In Haskell, we evaluate expressions when we *need* them rather than when they are first referred to or constructed. This is one of the ways in which non-strictness makes Haskell expressive—we can refer to values before we've done the work to create them.

This pattern applies to data structures and lambdas alike. You've already seen the effects of outside-in evaluation in the chapter on folds. Outside-in evaluation is why we can take the length of a list without touching any of the contents. Consider the following:

```
-- using an old definition of foldr
foldr k z xs = go xs
  where
    go []     = z
    go (y:ys) = y `k` go ys

c = foldr const 'z' ['a'..'e']
```

Expanding the `foldr` in `c`:

```
c = const 'z' "abcde" = go "abcde"
  where
    go []     = 'z'
    go ('a':"bcde") = 'a' `const` go "bcde"
```

So, the first step of evaluation of the fold here is:

```
const 'a' (go "bcde")
```

```
const  x       y      =  x
const 'a' (go "bcde") = 'a'
```

The second argument and step of the fold is never evaluated:

```
const 'a' _ = 'a'
```

It doesn't even matter if the next value is bottom:

```
Prelude> foldr const 'z' ['a', undefined]
'a'
```

This is outside-in showing itself. The `const` function was in the outermost position, so it was evaluated first.

27.4 What does the other way look like?

In strict languages, you *cannot* ordinarily bind a computation to a name without having already done all the work to construct it.

We'll use this example program to compare inside-out and outside-in (strict and non-strict) evaluation strategies:

```
module OutsideIn where

hypo :: IO ()
hypo = do
  let x :: Int
      x = undefined
  s <- getLine
  case s of
    "hi" -> print x
    _    -> putStrLn "hello"
```

For a strict language, this is a problem. A strict language cannot evaluate `hypo` successfully unless the `x` isn't bottom. This is because strict languages will force the bottom before binding `x`. A strict language is evaluating each binding as it comes into scope, not when a binding is used.

In non-strict Haskell, you can probably guess how this'll go:

```
Prelude> hypo
s
hello
Prelude> hypo
hi
*** Exception: Prelude.undefined
```

The idea is that evaluation is driven by demand, *not* by construction. We don't get the exception unless we force the evaluation of `x`—outside in.

27.5 Can we make Haskell strict?

Let's see if we can replicate the results of a strict language, though, which will give us a good picture of how Haskell is different. We can add strictness here in the following manner:

```
hypo' :: IO ()
hypo' = do
  let x :: Integer
      x = undefined
  s <- getLine
  case x `seq` s of
    "hi" -> print x
    _ -> putStrLn "hello"
```

Running it will give this result:

```
Prelude> hypo'
asd
*** Exception: Prelude.undefined
```

Why? Because this little `seq` function magically forces evaluation of the first argument if and when the second argument has to be evaluated. Adding `seq` means that anytime `s` is evaluated, `x` must also be evaluated. We'll get into more detail in a moment.

One thing to note before we investigate `seq` is that we manage to run `getLine` *before* the bottom gets evaluated, so this still isn't quite what a strict language would do. Case expressions are, in general, going to force evaluation. This makes sense if you realize it has to evaluate the expression to discriminate on the cases. A small example to demonstrate:

```
let b = ???
case b of
  True -> ...
  False
```

Here, `b` could be pretty much anything. It must evaluate `b` to find out if the expression results in `True` or `False`.

seq and ye shall find

Before we move any further with making Haskell stricter, let's talk about `seq` a little bit. One thing is that the type is, uh, a bit weird:

```
seq :: a -> b -> b
```

Clearly there's more going on here than `flip const`. It might help to know that in some old versions of Haskell, it used to have the type:

```
seq :: Eval a => a -> b -> b
```

`Eval` is short for *evaluation to weak head normal form*, and it provided a method for forcing evaluation. Instances were provided for all the types in `base`. It was elided in part so you could use `seq` in your code without churning your polymorphic type variables and forcing a bunch of changes. With respect to bottom, `seq` is defined as behaving in the following manner:

```
seq bottom b                    = bottom
seq literallyAnythingNotBottom b = b
```

Now, why does `seq` look like `const`'s gawky cousin? Since evaluation in Haskell is *demand driven*, we can't guarantee that something will *ever* be evaluated, *period*. Instead, we have to create links between nodes in the graph of expressions where forcing one expression will force yet another expression. Let's look at another example:

```
Prelude> :{
*Main| let wc x z =
*Main|       let y =
*Main|             undefined `seq` 'y' in x
*Main| :}
Prelude> foldr wc 'z' ['a'..'e']
'a'
Prelude> foldr (flip wc) 'z' ['a'..'e']
'z'
```

We never evaluate `y`, so we never force the bottom. However, we can lash yet another data dependency from `y` to `x`:

```
Prelude> bot = undefined
Prelude> :{
*Main| let wc x z =
*Main|       let y =
*Main|            bot `seq` 'y'
*Main|       in   y `seq` x
*Main| :}
Prelude> foldr wc 'z' ['a'..'e']
*** Exception: Prelude.undefined
Prelude> foldr (flip wc) 'z' ['a'..'e']
*** Exception: Prelude.undefined
```

Previously, the evaluation dependency was between the bottom value and `y`:

```
undefined `seq` y

-- forcing y necessarily forces undefined

y -> undefined
```

Changing the expression as we did causes the following to happen:

```
undefined `seq` y `seq` x

-- forcing x necessarily forces y
-- forcing y necessarily forces undefined

x -> y -> undefined
```

We can think of this as a chain reaction.

All we can do is chuck a life raft from one value to another as a means of saying, "If you want to get him, you gotta get through me!" We can even set our life-raft buddies adrift! Check it out:

```
notGonnaHappenBru :: Int
notGonnaHappenBru =
  let x = undefined
      y = 2
      z = (x `seq` y `seq` 10, 11)
  in snd z
```

The above will *not* bottom out! Our life-raft buddies are bobbing in the ocean blue, with no tugboat evaluator to pull them in.

seq and weak head normal form

What `seq` does is evaluate your expression *up to weak head normal form*. We've discussed it before, but if you'd like a deeper investigation and contrast of weak head normal form and normal form, we strongly recommend Simon Marlow's *Parallel and Concurrent Programming in Haskell*. WHNF evaluation means it stops at the first data constructor or lambda. Let's test that hypothesis!

```
Prelude> dc = (,) undefined undefined
Prelude> noDc = undefined
Prelude> lam = \_ -> undefined
Prelude> dc `seq` 1
1
Prelude> noDc `seq` 1
*** Exception: Prelude.undefined
```

```
Prelude> lam `seq` 1
1
```

Right-o. No surprises, right? Right? OK.

Since `dc` has a data constructor, `seq` doesn't need to care about the values inside that constructor—weak head normal form evaluation only requires it to evaluate the constructor. On the other hand, `noDc` has no data constructor or lambda outside the value, so there's no head for the evaluation to stop at. Finally, `lam` has a lambda outside the expression, which has the same effect on evaluation that a data constructor has: it stops it cold in its tracks.

Case matching also chains evaluation

This forcing behavior happens already without `seq`! For example, when you case or pattern match on something, you're forcing the value you pattern match on, because the compiler doesn't know which data constructor is relevant until it is evaluated to the depth required to yield the depth of data constructors on which you pattern matched. Let's look at an example:

```
data Test =
    A Test2
  | B Test2
  deriving (Show)

data Test2 =
    C Int
  | D Int
  deriving (Show)

forceNothing :: Test -> Int
forceNothing _ = 0

forceTest :: Test -> Int
forceTest (A _) = 1
forceTest (B _) = 2
```

```
forceTest2 :: Test -> Int
forceTest2 (A (C i)) = i
forceTest2 (B (C i)) = i
forceTest2 (A (D i)) = i
forceTest2 (B (D i)) = i
```

We'll test `forceNothing` first:

```
Prelude> forceNothing undefined
0
Prelude> forceNothing (A undefined)
0
```

It'll never bottom out, because it never forces anything. It's just a constant value that drops its argument on the floor. What about `forceTest`?

```
Prelude> forceTest (A undefined)
1
Prelude> forceTest (B undefined)
2
Prelude> forceTest undefined
*** Exception: Prelude.undefined
```

We only get a bottom when the outermost `Test` value is bottom, because that's the only value whose data constructors we're casing on. And then with `forceTest2`:

```
Prelude> forceTest2 (A (C 0))
0
Prelude> forceTest2 (A (C undefined))
*** Exception: Prelude.undefined

Prelude> forceTest2 (A undefined)
*** Exception: Prelude.undefined

Prelude> forceTest2 undefined
*** Exception: Prelude.undefined
```

There we go: outside -> in.

Core dump

Not the usual core dump you might be thinking of. In this case, we're talking about the underlying language that GHC Haskell gets simplified to after the compiler has desugared our code.

Our first means of determining strictness was by injecting bottoms into our expressions and observing the evaluation. Injecting bottoms everywhere allows us to see clearly what's being evaluated strictly and what's not. Our second means of determining strictness in Haskell is examining the GHC Core language.[3]

Here's the example we'll be working with:

```
module CoreDump where

discriminatory :: Bool -> Int
discriminatory b =
  case b of
    False -> 0
    True -> 1
```

Load this up in GHCi in the following manner:

```
Prelude> :set -ddump-simpl
Prelude> :l coreDump.hs
[1 of 1] Compiling CoreDump

================ Tidy Core ==============
...some noise...
```

You should then get the following GHC Core output:

[3] https://ghc.haskell.org/trac/ghc/wiki/Commentary/Compiler/CoreSynType

```
discriminatory :: Bool -> Int
[GblId, Arity=1,
 Caf=NoCafRefs,
 Str=DmdType]
discriminatory =
  \ (b_aZJ :: Bool) ->
    case b_aZJ of _ [Occ=Dead] {
      False -> GHC.Types.I# 0;
      True -> GHC.Types.I# 1
    }
```

We're not going to dissemble: GHC Core is a bit ugly. However, there are some means of cleaning it up. One is to use the `-dsuppress-all` flag:

```
Prelude> :set -dsuppress-all
Prelude> :r
```

Note that you may need to poke the file to force it to reload. This then outputs:

```
discriminatory
discriminatory =
  \ b_aZY ->
    case b_aZY of _ {
      False -> I# 0;
      True -> I# 1
    }
```

A titch more readable. The idea here is that the simpler Core language gives us a clearer idea of *when precisely* something will be evaluated. For the sake of simplicity, we'll revisit a previous example:

```
forceNothing _ = 0
```

In Core, it looks like this:

```
forceNothing = \ _ -> I# 0#
```

We're looking for case expressions in GHC Core to find out where the strictness is in our code, because case expressions must be evaluated. There aren't any cases here, so it forces strictly nothing! `I# 0#` is the underlying representation of an `Int` literal, which is exposed in GHC Core. On with the show!

Let's see what the Core for `forceTest` looks like:

```
forceTest =
  \ ds_d2oX ->
    case ds_d2oX of _ {
      A ds1_d2pI -> I# 1#;
      B ds1_d2pJ -> I# 2#
    }
```

From the GHC Core for this, we can see that we force one value, the outermost data constructors of the `Test` type. The contents of those data constructors are given a name but never used and so are never evaluated. Let's look at another example:

```
forceTest2 =
  \ ds_d2n2 ->
    case ds_d2n2 of _ {
      A ds1_d2oV ->
        case ds1_d2oV of _ {
          C i_a1lo -> i_a1lo;
          D i_a1lq -> i_a1lq
        };

      B ds1_d2oW ->
        case ds1_d2oW of _ {
          C i_a1lp -> i_a1lp;
          D i_a1lr -> i_a1lr
        }
    }
```

With `forceTest2`, the outsideness and insideness shows more clearly. In the outer part of the function, we do the same as `forceTest`, but the difference is that we also end up forcing the contents of the outer `Test` data constructors. This function has four possible results that

aren't bottom, and if bottom isn't passed to it, it will always force twice—once for `Test` and once for `Test2`. It returns but does not itself force or evaluate the contents of the `Test2` data constructor.

In Core, a case expression *always* evaluates what it cases on—even if no pattern matching is performed—whereas in Haskell proper, values are forced when matching on data constructors. We recommend reading the GHC documentation on the Core language referenced in the footnote above if you'd like to leverage Core to understand your Haskell code's performance or behavior more deeply.

Now, let us use this to analyze something:

```
discriminatory :: Bool -> Int
discriminatory b =

  let x = undefined
  in case b of
    False -> 0
    True -> 1
```

What does the Core for this look like?

```
discriminatory
discriminatory =

  \ b_a10c ->
    case b_a10c of _ {
      False -> I# 0;
      True -> I# 1
    }
```

GHC is too clever for our shenanigans! It knows we'll *never* evaluate `x`, so it drops it. What if we *force* it to evaluate `x` before we evaluate `b`?

```
discriminatory :: Bool -> Int
discriminatory b =
  let x = undefined
  in case x `seq` b of
    False -> 0
    True -> 1
```

Then the Core:

```
discriminatory =
  \ b_a10D ->
    let {
      x_a10E
      x_a10E = undefined } in
    case
      case x_a10E of _ {
        __DEFAULT -> b_a10D
      } of _ {
        False -> I# 0;
        True -> I# 1
    }
```

What's happened here is that there are now *two* case expressions, one nested inside another. The nesting is to make the evaluation of `x` obligatory before evaluating `b`. This is how `seq` changes your code.

A Core difference In Haskell, case matching is strict—or, at least, the pattern matching of it is—up to WHNF. In Core, cases are always strict to WHNF. This doesn't seem to be a distinction that matters, but there are times when it does become relevant. In Haskell, this will not bottom out:

```
case undefined of { _ -> False}
```

When the code above is translated into Core, the compiler recognizes that we didn't actually use the case match for anything and drops the case expression entirely, simplifying it to just the data constructor `False`.

However, while the following Core expression is syntactically similar to the normal Haskell code above, it will bottom out:

```
case undefined of { DEFAULT -> False }
```

Case in Core is strict even if there's one case and it doesn't match on anything. Core and Haskell are not the same language, but anytime you need to know if two expressions in Haskell are the same, one way to know for sure is by examining the Core.

A little bit stricter now

OK, we had a nice digression there into wonderland! Let's get back to the point, which is... we still haven't quite managed to accomplish what a strict language would have done with our `hypo` function, because we did partially evaluate the expression. We evaluated the `s`, which forced the `x`, which is what finally gave us the exception. A strict language would not even have evaluated `s`, because evaluating `s` would depend on the `x` inside already having been evaluated.

What if we want our Haskell program to do as a strict language would have done?

```
hypo'' :: IO ()
hypo'' = do

  let x :: Integer
      x = undefined
  s <- x `seq` getLine

  case s of
    "hi" -> print x
    _ -> putStrLn "hello"
```

Notice that we move the `seq` to the *earliest* possible point in our IO action. This one will just pop without so much as a by-your-leave:

```
Prelude> hypo''
*** Exception: Prelude.undefined
```

The reason is, we're forcing evaluation of the bottom *before* we evaluate `getLine`, which would have performed the effect of awaiting user input. While this reproduces the observable results of what a strict language might have done, it isn't truly the same thing, because we're not firing off the error on the construction of the bottom. It's not possible for an expression to be evaluated until the path that evaluation takes through your program has reached that expression. In Haskell, the tree doesn't fall in the woods until you walk through the forest and get to the tree. For that matter, the tree doesn't exist until you walk up to it.

Exercises: Evaluate

Expand the following expressions in as much detail as possible. Then, work outside-in to see what the expressions evaluate to:

1. `const 1 undefined`
2. `const undefined 1`
3. `flip const undefined 1`
4. `flip const 1 undefined`
5. `const undefined undefined`
6. `foldr const 'z' ['a'..'e']`
7. `foldr (flip const) 'z' ['a'..'e']`

27.6 Call by name, call by need

Another way we can talk about different evaluation strategies is by distinguishing them on the basis of call by name, call by need, and call by value.

1. Call by value: argument expressions are evaluated before entering a function. The expressions that bindings reference are evaluated before the bindings are created. This is conventionally called *strict*. This is inside-out evaluation.
2. Call by name: expressions can be arguments to a function without having been evaluated, or, in some cases, without ever being evaluated. You can create bindings to expressions without evaluating them first. Non-strictness includes this evaluation strategy. This is outside-in.
3. Call by need: this is the same as call by name, but expressions are only evaluated once. This only happens some of the time in GHC Haskell, usually when an expression isn't a lambda that takes arguments and also has a name. Results are typically shared within that name only in GHC Haskell (that is, other implementations of Haskell may choose to do things differently). This is also non-strict and outside-in.

27.7 Non-strict evaluation changes what we can do

We'll cover normal order evaluation (the non-strict strategy Haskell prescribes for its implementations) in more detail later. For now, let's look at some examples of what non-strictness enables. The following will work in languages with a strict or non-strict evaluation strategy:

```
Prelude> myList = [1, 2, 3]
Prelude> tail myList
[2,3]
```

This works in either strict or non-strict languages, because there is nothing there that *can't* be evaluated. However, if we keep in mind that `undefined` as an instance of bottom will throw an error when forced:

```
Prelude> undefined
*** Exception: Prelude.undefined
```

We'll see a difference between strict and non-strict. This will only work in languages that are non-strict:

```
Prelude> myList = [undefined, 2, 3]
Prelude> tail myList
[2,3]
```

A strict language would have crashed on construction of `myList` due to the presence of bottom. This is because strict languages eagerly evaluate all expressions as soon as they are constructed. The moment `[undefined, 2, 3]` is declared, `undefined` is evaluated as an argument to `:`, which raises the exception. In Haskell, however, non-strict evaluation means that bottom value won't be evaluated unless it is needed for some reason.

Take a look at the next example, and, before going on, see if you can figure out whether it will throw an exception and why:

```
Prelude> head $ sort [1, 2, 3, undefined]
```

When we call `head` on a list that has been passed to `sort`, we only need the lowest value in the list, and that's all the work we will do. The problem is that in order for `sort` to know what the lowest value is, it must evaluate `undefined`, which then throws the error.

27.8 Thunk Life

A *thunk* is used to reference suspended computations that might be performed or computed at a later point in your program. We could delve into considerably more detail on this topic,[4] but essentially thunks are computations that are not yet evaluated up to weak head normal form. If you read the GHC notes on this, you'll see references to *head normal form*—it's the same thing as weak head normal form.

Not all values get thunked

We're going to be using the GHCi command `sprint` in this section as one means of showing when something is thunked. You may remember this from Chapter 9, on lists, but let's refresh our memories.

The `sprint` command allows us to show what has been evaluated already by printing in the REPL. An underscore is used to represent values that haven't been evaluated yet. We noted before that this command can have some quirky behavior, although this chapter will explain some of the things that cause those seemingly unpredictable behaviors.

Let's start with a simple example:

```
Prelude> myList = [1, 2] :: [Integer]
Prelude> :sprint myList
myList = [1,2]
```

Wait a second—what happened here? Why is the list shown fully evaluated when it's not been needed by anything? This is opportunistic strictness. GHC will not thunk (and thus delay) data constructors. Data constructors are known to be constant, which justifies the safety of the optimization. The data constructors here are cons, or `:`, the `Integer` values, and the empty list—all of them are constants.

But aren't data constructors functions? Data constructors are like functions when they're unapplied and like constants once they are fully applied. Since all the data constructors in the above example are fully applied already, evaluating to weak head normal form means evaluating everything, because there's nothing left to apply.

[4] `https://ghc.haskell.org/trac/ghc/wiki/Commentary/Rts/Storage/HeapObjects`

Now back to the thunkery.

A graph of the values of `myList` looks like this:

Here, there aren't any unevaluated thunks—it's just the final values that, once calculated, the program remembers. However, if we make it more polymorphic:

```
Prelude> myList2 = [1, 2, 3]
Prelude> :t myList2
myList2 :: Num a => [a]
Prelude> :sprint myList2
myList2 = _
```

We'll see an unevaluated thunk represented by the underscore at the very top level of the expression. Since the type is not concrete, there's an implicit function `Num a -> a` underneath, awaiting application to something that will force it to evaluate to a concrete type. There's nothing here triggering that evaluation, so the whole list remains an unevaluated thunk. We'll get into more detail about how type class constraints evaluate soon.

GHC will also stop opportunistically evaluating as soon as it hits a computation:

```
Prelude> myList = [1, 2, id 1] :: [Integer]
Prelude> :sprint myList
myList = [1,2,_]
```

It's a trivial computation, but GHCi conveniently leaves it be. Here's the thunk graph for the above:

Now, let us consider another case that might be initially slightly confusing for some:

```
Prelude> myList = [1, 2, id 1] :: [Integer]
Prelude> myList' = myList ++ undefined
Prelude> :sprint myList'
myList' = _
```

Whoa, whoa, whoa. What's going on here? The whole thing is thunked, because it's not in weak head normal form. Why isn't it in weak head normal form, already? Because the outermost term isn't a data constructor like `:` is. The outermost term is the *function* `++`:

```
myList' = (++) _ _
```

The function is outermost, despite the fact that it is superficially an infix operator, because the function is the lambda. The arguments are passed into the function body to be evaluated later.

27.9 Sharing is caring

Sharing here roughly means what we've implied above: that when a computation is named, the results of evaluating that computation can be shared between all references to that name without re-evaluating it. We care about sharing, because memory is finite, even today in the land of chickens in every pot and smartphones in every pocket. The idea here is that non-strictness is a fine thing, but call-by-name semantics aren't always enough to make it sufficiently efficient. What is sufficiently efficient? That depends on context and whether it's your dissertation or not.

One of the points of confusion for people when trying to figure out how GHC Haskell really runs code is that it turns sharing on and off (that is, it oscillates between call-by-need and call-by-name) based on necessity and what it thinks will produce faster code. Part of the reason it can do this at all without breaking your code is because the compiler knows when your code does or does not perform I/O.

Using trace to observe sharing

The *base* library has a module named `Debug.Trace` that has functions useful for observing sharing. We'll mostly use `trace` here, but feel free to poke around for whatever else might catch your fancy. `Debug.Trace` is a means of cheating the type system and inserting a `putStrLn` into your code without having `IO` in the type. This is *definitely* something you want to restrict to experimentation and education; do not use it as a logging mechanism in production code—it won't do what you think. However, it does give us a convenient means of observing when things evaluate.

Let us demonstrate how we can use this to see when things get evaluated:

```
Prelude> import Debug.Trace
Prelude> a = trace "a" 1
Prelude> b = trace "b" 2
Prelude> a + b
b
a
3
```

This isn't an example of sharing, but it demonstrates how `trace` can be used to observe evaluation. We can see that `b` is printed first, because that is the first argument that the addition function evaluates, but you cannot and should not rely on the evaluation order of the arguments to addition. Here, we're talking about the order in which the arguments to a single application of addition are forced, not associativity. You can count on addition being left associative, but within each pairing, which in the pair of arguments gets forced is not guaranteed.

Let's look at a longer example and see how it shows us where the evaluations occur:

```
import Debug.Trace (trace)

inc = (+1)

twice = inc . inc

howManyTimes =
  inc (trace "I got eval'd" (1 + 1))
    + twice
      (trace "I got eval'd" (1 + 1))

howManyTimes' =
  let onePlusOne =
        trace "I got eval'd" (1 + 1)
  in inc onePlusOne + twice onePlusOne
```

```
Prelude> howManyTimes
I got eval'd
I got eval'd
7

Prelude> howManyTimes'
I got eval'd
7
```

Cool, with that in mind, let's talk about ways to promote and prevent sharing.

What promotes sharing

Kindness. Also, names. Names turn out to be a pretty good way to make GHC share something, *if* it could have otherwise been shared. First, let's consider the example of something that won't get shared:

```
Prelude> import Debug.Trace
Prelude> x = trace "x" (1 :: Int)
Prelude> y = trace "y" (1 :: Int)
Prelude> x + y
x
```

```
y
2
```

This seems intuitive and reasonable, but the values of `x` and `y` cannot be shared, because they have different names. So, even though they have the same value, they have to be evaluated separately.

GHC does use this intuition, that you expect results to be shared when they have the same name, to make performance more predictable. If we add two values that have the same name, that named value will get evaluated once and only once:

```
Prelude> import Debug.Trace
Prelude> a = trace "a" (1 :: Int)
Prelude> a + a
a
2
Prelude> a + a
2
```

Indirection won't change this, either:

```
Prelude> x = trace "x" (1 :: Int)
Prelude> (id x) + (id x)
x
2
Prelude> (id x) + (id x)
2
```

GHC knows what's up, despite the addition of identity functions. Notice that the second time we run it, it doesn't evaluate `x` at all. The value of `x` is held in memory, so whenever your program calls `x`, it already knows the value.

In general, as mentioned above, GHC relies on an intuition around names and sharing to make performance more predictable. However, this won't always behave in ways you expect. Consider the case of a list with a single character... and a `String` with a single character. They're actually the same thing, but the way they get constructed is not. This produces differences in the opportunistic strictness GHC will engage in:

```
Prelude> a = Just ['a']
Prelude> :sprint a
a = Just "a"

Prelude> a = Just "a"
Prelude> :sprint a
a = Just _
```

So uh, what gives? Well, the deal is that the strictness analysis GHC performs here is limited to data constructors only, no computation! But where's the function you ask? Well, if we turn on our night vision goggles:

```
Prelude> a = Just ['a']

returnIO
  (: ((Just (: (C# 'a') ([])))
   `cast` ...) ([]))

Prelude> a = Just "a"

returnIO
  (: ((Just (unpackCString# "a"#))
   `cast` ...) ([]))
```

The issue is that a call to a primitive function in `GHC.Base` interposes between `Just` and a `CString` literal. The reason string literals aren't actually lists of characters at time of construction is mostly to present optimization opportunities, such as when we convert string literals into `ByteString` or `Text` values. More on that in the next chapter!

What subverts or prevents sharing

Sometimes, we don't want sharing. Sometimes, we want to know why sharing didn't happen when we did want it. Understanding what kinds of things prevent sharing is, therefore, useful.

Inlining expressions where they get used prevents sharing, because doing so creates independent thunks that will get computed separately. In this example, instead of declaring the value of `f` to equal 1, we make it a function:

```
Prelude> :{
Prelude| let f :: a -> Int
Prelude|     f _ = trace "f" 1
Prelude| :}
Prelude> f 'a'
f
1
Prelude> f 'a'
f
1
```

In the next examples, you can directly compare the difference between assigning a name to the value of `(2 + 2)` versus inlining it directly. When it's named, it gets shared and not re-evaluated:

```
Prelude> a :: Int; a = trace "a" 2 + 2
Prelude> b = (a + a)
Prelude> b
a
8
Prelude> b
8
```

Here, we see `a` once, which makes sense, as we expect the result to be shared:

```
Prelude> :{
Prelude| let c :: Int;
Prelude|     c =   (trace "a" 2 + 2)
Prelude|         + (trace "a" 2 + 2)
Prelude| :}
Prelude> c
a
a
8
Prelude> c
8
```

Here, an expression equivalent to `a` isn't shared, because the two occurrences of the expression aren't bound to the same name. This

is a trivial example of inlining, but it illustrates the difference in how things evaluate when an expression is bound to a name versus when it gets repeated via inlining in an expression.

Being a function with explicit, named arguments also prevents sharing. Haskell is not fully lazy; it is merely non-strict, so it is not required to remember the result of every function application for a given set of arguments, nor would it be desirable given memory constraints. A demonstration:

```
Prelude> :{
Prelude| let f :: a -> Int
Prelude|     f = trace "f" const 1
Prelude| :}
Prelude> f 'a'
f
1
Prelude> f 'a'
1
Prelude> f 'b'
1
```

The explicit, named arguments part here is critical! Eta reduction (i.e., writing point-free code, thus dropping the named arguments) will change the sharing properties of your code. This will be explained in more detail in the next chapter.

Type class constraints also prevent sharing. If we forget to add a concrete type to an earlier example, we evaluate `a` twice:

```
Prelude> blah = Just 1
Prelude> fmap ((+1) :: Int -> Int) blah
Just 2
Prelude> :sprint blah
blah = _
Prelude> :t blah
blah :: Num a => Maybe a

Prelude> bl = Just 1
Prelude> :t bl
bl :: Num a => Maybe a
```

```
Prelude> :sprint bl
bl = _
Prelude> fmap (+1) bl
Just 2

Prelude> fm = fmap (+1) bl
Prelude> :t fm
fm :: Num b => Maybe b
Prelude> :sprint fm
fm = _
Prelude> fm
Just 2
Prelude> :sprint fm
fm = _

Prelude> :{
Prelude| let bla =
Prelude|   Just $ trace "eval'd 1" 1
Prelude| let fm' =
Prelude|   fmap ((+1) :: Int -> Int) bla
Prelude| :}
Prelude> fm'
Just eval'd 1
2
Prelude> :sprint fm'
fm' = Just 2
```

Again, that's because type class constraints are functions in Core. They are awaiting application to something that will make them become concrete types. We're going to go into a bit more detail on this in the next section.

Implicit parameters are implemented similarly to type class constraints and have the same effect on sharing. Sharing doesn't work in the presence of constraints (type classes or implicit parameters), because type class constraints and implicit parameters decay into function arguments when the compiler simplifies the code:

```
-- MyTrace.hs
{-# LANGUAGE ImplicitParams #-}

module MyTrace where

import Debug.Trace

add :: (?x :: Int) => Int
add = trace "add" 1 + ?x
```

```
Prelude> :set -XImplicitParams
Prelude> :l MyTrace.hs
[1 of 1] Compiling MyTrace
Ok, one module loaded.
Prelude> let ?x = 1 in add
add
2
Prelude> let ?x = 1 in add
add
2
```

We won't talk about implicit parameters too much more, as we don't think they're a good idea for general use. In most cases where you believe you want implicit parameters, more likely you want `Reader`, `ReaderT`, or a plain old function argument.

Why polymorphic values never seem to get forced

As we've said, GHC engages in opportunistic strictness when it can do so safely without making an otherwise valid expression result in bottom. This is one of the things that confounds the use of `sprint` for observing evaluation in GHCi—GHC will often be opportunistically strict with data constructors if it knows the contents definitely can't be a bottom, such as when they're a literal value. It gets more complicated when we consider that, under the hood, type class constraints are simplified into additional arguments.

Reusing a similar example from earlier, we will first observe this in action, and then we'll talk about why it happens:

```
Prelude> :{
Prelude| let blah =
Prelude|   Just (trace "eval'd 1" 1)
Prelude| :}
Prelude> :sprint blah
blah = _
Prelude> :t blah
blah :: Num a => Maybe a
Prelude> fmap (+1) blah
Just eval'd 1
2
Prelude> fmap (+1) blah
Just eval'd 1
2
Prelude> :sprint blah
blah = _
```

So we have at least some evidence that we're re-evaluating. Does it change when it's concrete?

```
Prelude> :{
Prelude| let blah =
Prelude|      Just (trace "eval'd 1"
Prelude|            (1 :: Int))
Prelude| :}
Prelude> :sprint blah
blah = Just _
```

The `Int` value being obscured by `trace` prevents opportunistic evaluation there. However, eliding the `Num a => a` in favor of a concrete type does bring sharing back:

```
Prelude> fmap (+1) blah
Just eval'd 1
2
Prelude> fmap (+1) blah
Just 2
```

Now, our trace gets emitted only once. The idea here is that after the type class constraints get simplified to the underlying GHC Core language, they're really function arguments.

It doesn't matter if you use a function that accepts a concrete type and forces the `Num a => a`, it'll redo the work on each evaluation because of the type class constraint. For example:

```
Prelude> fmap ((+1) :: Int -> Int) blah
Just 2
Prelude> :sprint blah
blah = _
Prelude> :t blah
blah :: Num a => Maybe a

Prelude> bl = Just 1
Prelude> :t bl
bl :: Num a => Maybe a
Prelude> :sprint bl
bl = _
Prelude> fmap (+1) bl
Just 2
Prelude> fm = fmap (+1) bl
Prelude> :t fm
fm :: Num b => Maybe b
Prelude> :sprint fm
fm = _
Prelude> fm
Just 2
Prelude> :sprint fm
fm = _

Prelude> :{
Prelude| let fm' =
Prelude|       fmap ((+1) :: Int -> Int)
Prelude|            blah
Prelude| :}
Prelude> fm'
Just eval'd 1
2
Prelude> :sprint fm'
fm' = Just 2
```

So, what's the deal here with the type class constraints? It's as if `Num a => a` were really `Num a -> a`. In Core, they are. The only way to apply that function argument is to reach an expression that provides a concrete type satisfying the constraint. Here's a demonstration of the difference in behavior using values:

```
Prelude> poly = 1
Prelude> conc = poly :: Int
Prelude> :sprint poly
poly = _
Prelude> :sprint conc
conc = _
Prelude> poly
1
Prelude> conc
1
Prelude> :sprint poly
poly = _
Prelude> :sprint conc
conc = 1
```

`Num a => a` is a function awaiting an argument, while `Int` is not. Behold the Core:

```
module Blah where

a :: Num a => a
a = 1

concrete :: Int
concrete = 1
```

```
Prelude> :l code/blah.hs
[1 of 1] Compiling Blah

================ Tidy Core ==============
Result size of Tidy Core =
  {terms: 9, types: 9, coercions: 0}

concrete
```

```
concrete = I# 1


a
a =
  \ @ a1_aRN $dNum_aRP ->
    fromInteger $dNum_aRP (__integer 1)
```

Do you see how `a` has a lambda? In order to know what instance of the type class to deploy at any given time, the type has to be concrete. As we've seen, types can become concrete through assignment or type defaulting. Whichever way it becomes concrete, the result is the same: once the concrete type is known, the type class constraint function gets applied to the type class instance for that type. If you don't declare the concrete type, it will have to re-evaluate this function every time, because it can't know that the type didn't change somewhere along the way. So, because it remains a function and unapplied functions are not shareable values, polymorphic expressions can't be shared.

Mostly, the behavior doesn't change when it involves values defined in terms of functions, but if you forget the type concretion, it'll stay _, and you'll be confused and upset. Observe:

```
Prelude> :{
Prelude| let blah :: Int -> Int
Prelude|     blah x = x + 1
Prelude| :}
Prelude> woot = blah 1
Prelude> :sprint blah
blah = _
Prelude> :sprint woot
woot = _
Prelude> woot
2
Prelude> :sprint woot
woot = 2
```

Values of a concrete, constant type can be shared, once evaluated. Polymorphic values may be evaluated once but still not shared, because, underneath, they continue to be functions awaiting application.

Preventing sharing on purpose

When do we *want* to prevent sharing? When we don't want a large datum hanging out in memory that was calculated to provide a much smaller answer. First an example that demonstrates sharing:

```
Prelude> import Debug.Trace
Prelude> f x = x + x
Prelude> f (trace "hi" 2)
hi
4
```

We see ``hi'' once, because x gets evaluated once. In the next example, x gets evaluated twice:

```
Prelude> f x = (x ()) + (x ())
Prelude> f (\_ -> trace "hi" 2)
hi
hi
4
```

Using unit, (), as the argument to x turns x into a trivial, weird-looking function, which is why the value of x can no longer be shared. It doesn't matter much since that "function" x doesn't really *do* anything.

OK, that was strange. Maybe it'll be easier to see the point here if we give a more traditional-seeming argument to x:

```
Prelude> f x = (x 2) + (x 10)
Prelude> f (\x -> trace "hi" (x + 1))
hi
hi
14
```

Using a lambda that binds the argument in some fashion disables sharing:

```
Prelude> g = \_ -> trace "hi" 2
Prelude> f g
hi
hi
4
```

However, this works in part, because the function passed to `f` has the argument as part of the declaration, even though it uses an underscore to ignore it. Notice what happens if we make it point-free:

```
Prelude> g = const (trace "hi" 2)
Prelude> f g
hi
4
```

We're going to get into a little more detail about this distinction in the next chapter, but the idea here is that functions aren't shared when there are named arguments but are when the arguments are elided, as in point-free style. So, one way to prevent sharing is to add named arguments.

Forcing sharing

You can force sharing by giving your expression a name. The most common way of doing this is with `let`:

```
-- calculates 1 + 1 twice
(1 + 1) * (1 + 1)

-- shares 1 + 1 result under x
let x = 1 + 1
in x * x
```

With that in mind, if you take a look at the `forever` function in `Control.Monad`, you might see something a little mysterious looking:

```
forever     :: (Monad m) => m a -> m b
forever a   = let a' = a >> a' in a'
```

Why the `let` expression? Well, we want sharing here so that running a monadic action indefinitely doesn't leak memory. The sharing here causes GHC to overwrite the thunk as it runs each step in the evaluation, which is quite handy. Otherwise, it would keep constructing new thunks indefinitely, and that would be most unfortunate.

27.10 Refutable and irrefutable patterns

When we're talking about pattern matching, it's important to be aware that there are refutable and irrefutable patterns. An *irrefutable pattern* is one that will never fail to match. A *refutable pattern* is one that does have potential failures. Often, the problem is one of specificity:

```
refutable :: Bool -> Bool
refutable True = False
refutable False = True

irrefutable :: Bool -> Bool
irrefutable x = not x

oneOfEach :: Bool -> Bool
oneOfEach True = False
oneOfEach _ = True
```

Remember, the *pattern* is refutable or not, not the function itself. The function `refutable` is refutable, because each case is refutable; each case could be given an input that fails to match. In contrast, `irrefutable` has an irrefutable pattern; that is, its pattern doesn't rely on matching with a specific value.

In the case of `oneOfEach`, the first pattern is refutable, because it pattern matches on the `True` data constructor. `irrefutable` and the second match of `oneOfEach` are irrefutable, because they don't need to look inside the data they are applied to.

That said, the second pattern match of `oneOfEach` being irrefutable isn't terribly meaningful, in terms of semantics, as Haskell will have to inspect the data to determine whether it matches the first case anyway.

The `irrefutable` function works for *any* inhabitant (all two of them) of `Bool`, because it doesn't specify which `Bool` value in the pattern to match. You could think of an *irrefutable* pattern as one that will never fail to match. If an irrefutable pattern for a particular value comes before a refutable pattern, the refutable pattern will never get invoked.

This little function appeared in an earlier chapter, but we'll bring it back for a quick demonstration:

```
isItTwo :: Integer -> Bool
isItTwo 2 = True
isItTwo _ = False
```

In the case of `Bool`, the order of matching `True` and `False` specifically doesn't matter, but in cases like `isItTwo`, where one case is specific and the other is a catch-all, otherwise case, the ordering will certainly matter. You can reorder the expressions of `isItTwo` to see what happens, although it's probably clear.

Lazy patterns

Lazy patterns are also *irrefutable*.

```
strictPattern :: (a, b) -> String
strictPattern (a,b) = const "Cousin It" a

lazyPattern :: (a, b) -> String
lazyPattern ~(a,b) = const "Cousin It" a
```

The tilde is how one makes a pattern match lazy. A caveat is that since it makes the pattern irrefutable, you can't use it to discriminate cases of a sum—it's useful instead for unpacking products that might not get used:

```
Prelude> strictPattern undefined
*** Exception: Prelude.undefined

Prelude> lazyPattern undefined
"Cousin It"
```

And as we see here, in the lazy pattern version, since `const` doesn't actually need `a` from the tuple, we never force the bottom. The default behavior is to just go ahead and force it before evaluating the function body, mostly for more predictable memory usage and performance.

27.11 Bang patterns

Sometimes, we want to evaluate an argument to a function whether we use it or not. We can do this with `seq`, as in the following example:

```
{-# LANGUAGE BangPatterns #-}

module ManualBang where

doesntEval :: Bool -> Int
doesntEval b = 1

manualSeq :: Bool -> Int
manualSeq b = b `seq` 1
```

Or, we can also do it with a bang pattern on `b`—note the exclamation point:

```
banging :: Bool -> Int
banging !b = 1
```

Let's look at the Core for those three:

```
doesntEval
doesntEval =
  \ _ -> I# 1#

manualSeq
manualSeq =
  \ b_a1ia ->
    case b_a1ia of _
      { __DEFAULT -> I# 1# }

banging
banging =
  \ b_a1ib ->
    case b_a1ib of _
      { __DEFAULT -> I# 1# }
```

If you try passing bottom to each function, you'll find that `manualSeq` and `banging` are forcing their arguments despite not using them for anything. Remember that forcing something is expressed in Core as a case and that case expressions evaluate up to weak head normal form in the Core language.

Bang patterns in data

When we evaluate the outer data constructor of a datatype, at times we'd also like to evaluate the contents to weak head normal form, just like we do with functions.

One way to see the difference between strict and non-strict constructor arguments is how they behave when they are undefined. Let's look at an example (note the exclamation mark):

```
data Foo = Foo Int !Int

first (Foo x _) = x
second (Foo _ y) = y
```

Since the non-strict argument isn't evaluated by `second`, passing in `undefined` doesn't cause a problem:

```
> second (Foo undefined 1)
1
```

But the strict argument can't be undefined, even if we don't use the value:

```
> first (Foo 1 undefined)
*** Exception: Prelude.undefined
```

You could do this manually with `seq`, but it's a little tedious. Here's another example with two equivalent datatypes, one of them with strictness annotations on the contents and one without:

```
{-# LANGUAGE BangPatterns #-}

module ManualBang where

data DoesntForce =
  TisLazy Int String

gibString :: DoesntForce -> String
gibString (TisLazy _ s) = s

-- note the exclamation marks again
data BangBang =
  SheShotMeDown !Int !String
```

```
gimmeString :: BangBang -> String
gimmeString (SheShotMeDown _ s) = s
```

Then, testing those in GHCi:

```
Prelude> x = TisLazy undefined "blah"
Prelude> gibString x
"blah"
Prelude> s = SheShotMeDown
Prelude> x = s undefined "blah"
Prelude> gimmeString x
"*** Exception: Prelude.undefined
```

The idea here is that in some cases, it's cheaper to just compute something than to construct a thunk and then evaluate it later. This case is particularly common in numerics code, where you have a lot of `Int` and `Double` values running around that are individually cheap to conjure. If the values are both cheap to compute and small, then you may as well make them strict unless you're trying to dance around bottoms. Types with underlying primitive representations like `Int` and `Double` most assuredly qualify as small.

A good rule to follow is lazy in the spine, strict in the leaves! Sometimes a "leak" isn't really a leak but temporarily excessive memory that subsides, because you made 1,000,000 tiny values into less-tiny thunks when you could have just computed them as your algorithm progressed.

27.12 Strict and StrictData

If you're using GHC 8.0 or newer, you can avail yourself of the `Strict` and `StrictData` extensions. The key thing to realize is that `Strict` and `StrictData` are just letting you avoid putting in pervasive uses of `seq` and bang patterns yourself. They don't add anything to the semantics of the language. Accordingly, they won't suddenly make lazy data structures defined elsewhere behave differently, although they do make functions for processing lazy data structures defined in modules that use them behave differently.

Let's play with them (if you have GHC 8.0 or newer; if not, this code won't work):

```
{-# LANGUAGE Strict #-}

module StrictTest where

blah x = 1

main = print (blah undefined)
```

The above will bottom out, because `blah` is defined under the module with the `Strict` extension and will get translated into the following:

```
blah x = x `seq` 1

-- or with bang patterns
blah !x = 1
```

So, the `Strict` and `StrictData` extensions are a means of avoiding noise when everything or almost everything in a module is supposed to be strict. You can use the tilde for irrefutable patterns in order to recover laziness on a case by case basis:

```
{-# LANGUAGE Strict #-}

module LazyInHostileTerritory where

willForce x = 1
willNotForce ~x = 1
```

Admittedly, these are glorified renames of `const`, but it doesn't matter for the purposes of demonstrating what happens. Here's what we see in GHCi when we pass bottom to the example functions above:

```
Prelude> willForce undefined
*** Exception: Prelude.undefined
Prelude> willNotForce undefined
1
```

So, even when you're using the `Strict` extension, you can selectively recover laziness when desired.

27.13 Adding strictness

Now, we will examine how applying strictness to a datatype and operations we're already familiar with can change how they behave in the presence of bottom through the list type. This is intended to be mostly demonstrative rather than a practical example:

```
module StrictTest1 where

data List a =
    Nil
  | Cons a (List a) deriving Show

sTake :: Int -> List a -> List a
sTake n _
  | n <= 0 = Nil
sTake n Nil = Nil
sTake n (Cons x xs) =
  (Cons x (sTake (n-1) xs))

twoEls = Cons 1 (Cons undefined Nil)
oneEl  = sTake 1 twoEls
```

The name of this module is a bit of a misnomer. `List` here is lazy, just like the built-in list type, `[]`, in `Prelude`. Our `take` derivative, named `sTake`, is also lazy.

Let's load up this code in our REPL and test it out:

```
Prelude> twoEls
Cons 1 (Cons
  *** Exception: Prelude.undefined

Prelude> oneEl
Cons 1 Nil
```

Now, let's experiment with adding strictness to different parts of our program and observe what changes in our code's behavior.

First, we're going to add `BangPatterns` so that we have a syntactically convenient way to denote when and where we want strictness:

```
module StrictTest2 where

data List a =
    Nil
  | Cons !a (List a) deriving Show

sTake :: Int -> List a -> List a
sTake n _
  | n <= 0 = Nil
sTake n Nil = Nil
sTake n (Cons x xs) =
  (Cons x (sTake (n-1) xs))

twoEls = Cons 1 (Cons undefined Nil)
oneEl  = sTake 1 twoEls
```

Noting the placement of the exclamation marks denoting strictness, let's run it in GHCi and see if it does what we want:

```
Prelude> twoEls
Cons 1 *** Exception: Prelude.undefined
Prelude> oneEl
Cons 1 Nil
```

Now, let's try a variation where the list type is strict in the value but not in the spine:

```
{-# LANGUAGE BangPatterns #-}

module StrictTest3 where

data List a =
    Nil
  | Cons !a (List a) deriving Show

sTake :: Int -> List a -> List a
sTake n _
  | n <= 0 = Nil
sTake n Nil = Nil
sTake n (Cons x !xs) =
  (Cons x (sTake (n-1) xs))
```

```
twoEls = Cons 1 (Cons undefined Nil)
oneEl  = sTake 1 twoEls

threeElements = Cons 2 twoEls
oneElT   = sTake 1 threeElements
```

We add strictness to the `xs` so that `sTake` forces more of the list. Let's see what happens:

```
Prelude> twoEls
Cons 1 *** Exception: Prelude.undefined

Prelude> oneEl
*** Exception: Prelude.undefined

Prelude> threeElements
Cons 2 (Cons 1
  *** Exception: Prelude.undefined

Prelude> oneElT
Cons 2 Nil
```

Let's add more strictness:

```
module StrictTest4 where

data List a =
    Nil
  | Cons !a !(List a) deriving Show

sTake :: Int -> List a -> List a
sTake n _
  | n <= 0 = Nil
sTake n Nil = Nil
sTake n (Cons x xs) =
  (Cons x (sTake (n-1) xs))

twoEls = Cons 1 (Cons undefined Nil)
oneEl  = sTake 1 twoEls
```

And run it again:

```
Prelude> twoEls
*** Exception: Prelude.undefined

Prelude> oneEl
*** Exception: Prelude.undefined
```

So, what's the upshot of our experiments with adding strictness here?

N	Cons	sTake
1	Cons a (List a)	Cons x xs
2	Cons !a (List a)	Cons x xs
3	Cons !a (List a)	Cons x !xs
4	Cons !a !(List a)	Cons x xs

Then the results themselves:

N	twoEls	oneEl
1	Cons 1 (Cons ***	Cons 1 Nil
2	Cons 1 ***	Cons 1 Nil
3	Cons 1 ***	***
4	***	***

You can see clearly what adding strictness in different places does to our evaluation in terms of bottom.

27.14 Chapter exercises

Strict list

Try messing around with the following list type and compare what it does with the bang-patterned list variants we experimented with earlier:

```
{-# LANGUAGE Strict #-}

module StrictList where
```

```
data List a =
  Nil |
  Cons a (List a)
  deriving (Show)

take' n _      | n <= 0 = Nil
take' _ Nil            = Nil
take' n (Cons x xs)    =
  (Cons x (take' (n-1) xs))

map' _ Nil         = Nil
map' f (Cons x xs) =
  (Cons (f x) (map' f xs))

repeat' x = xs where xs = (Cons x xs)

main = do
  print $ take' 10 $ map' (+1) (repeat' 1)
```

What will sprint output?

We show you a definition or multiple definitions, you determine what the `sprint` command will output when passed the bindings listed. Do each exercise in your head before testing it:

1. ```
   let x = 1
   ```

2. ```
   let x = ['1']
   ```

3. ```
   let x = [1]
   ```

4. ```
   let x = 1 :: Int
   ```

5. ```
   let f = \x -> x
   let x = f 1
   ```

6. ```
   let f :: Int -> Int; f = \x -> x
   let x = f 1
   ```

Will printing this expression result in bottom?

Same game, but this time try and determine whether or not each expression will bottom out:

1. ```
   snd (undefined, 1)
   ```

2. ```
   let x = undefined
   let y = x `seq` 1 in snd (x, y)
   ```

3. ```
   length $ [1..5] ++ undefined
   ```

4. ```
   length $ [1..5] ++ [undefined]
   ```

5. ```
   const 1 undefined
   ```

6. ```
   const 1 (undefined `seq` 1)
   ```

7. ```
   const undefined 1
   ```

Make the expression bottom

Using only bang patterns or `seq`, make the code bottom out when executed:

1. ```
   x = undefined
   y = "blah"
   main = do
     print (snd (x, y))
   ```

27.15 Follow-up resources

1. Simon Marlow. *Parallel and Concurrent Programming in Haskell*. See Chapter 2, "Basic Parallelism: The Eval Monad." https://www.oreilly.com/library/view/parallel-and-concurrent/9781449335939/ch02.html

2. Takenobu Tani. *Lazy evaluation illustrated for Haskell divers*. https://takenobu-hs.github.io/downloads/haskell_lazy_evaluation.pdf

3. John Launchbury. *A Natural Semantics for Lazy Evaluation*. https://galois.com/wp-content/uploads/2014/08/pub_JL_NaturalSemanticsForLazyEvaluation.pdf

4. Clem Baker-Finch, David King, Jon Hall, and Phil Trinder. *An Operational Semantics for Parallel Call-by-Need*. https://www.researchgate.net/publication/221241262_An_operational_semantics_for_parallel_lazy_evaluation

Chapter 28

Basic Libraries

> Bad programmers worry about the code. Good programmers worry about data structures and their relationships.
>
> Linus Torvalds

28.1 Basic libraries and data structures

Data structures are kind of important. Insofar as computers are fast, they aren't getting much faster—at least, the CPU isn't. This is usually a lead-in for a parallelism/concurrency sales pitch. But this isn't that book.

The data structures you choose to represent your problem affect the speed and memory involved in processing your data, perhaps to a larger extent than is immediately obvious. At the level of your program, making the right decision about how to represent your data is the first important step in writing efficient programs. In fact, your choice of data structure can affect whether it's worthwhile or it even makes sense to attempt to parallelize something.

This chapter is here to help you make decisions about the optimal data structures for your programs. We can't prescribe one or another of similar data structures, because how effective they are will depend a lot on what you're trying to do. So, our first step will be to give you tools to measure for yourself how different structures will perform in a given context. We'll also cover some of the mistakes that can cause your memory usage and execution time to explode.

This chapter will:

- Demonstrate how to measure the usage of time and space in your programs.
- Offer guidelines on when weak head normal form or normal form are appropriate when benchmarking code.
- Define constant applicative forms and explain argument saturation.
- Demonstrate and critically evaluate when to use different data structures in different circumstances.
- Sacrifice some jargon to the jargon gods.

We're going to kick this chapter off with some benchmarking.

28.2 Benchmarking with Criterion

It's a common enough thing to want to know how fast our code is. If you can't benchmark properly, then you can't know whether your

program took six microseconds to run or only five, and you can only ask yourself, "Well, do I feel lucky?"

Well, do ya, punk?

If you'd rather not trust your guesswork, the best way to measure performance is to sample many times in order to establish a confidence interval. Fortunately, that work has already been done for us in the wonderful library `criterion`[1] by Bryan O'Sullivan.

As it happens, `criterion` comes with a pretty nice tutorial,[2] but we'll still work through an example, so you can follow along with this chapter. In our toy program here, we're looking to write a total version of `!!`, which returns a `Maybe` in order to make bottoms unnecessary. When you compile code for benchmarking, make sure you're using `-O` or `-O2` in the build flags to GHC. Those can be specified by running GHC manually:

```
-- with stack
$ stack ghc -- -O2 bench.hs

-- without stack
$ ghc -O2 bench.hs
```

Or via the Cabal setting `ghc-options`.[3]

Let's get our module set up:

```
module Main where

import Criterion.Main

infixl 9 !?
_      !? n | n < 0 = Nothing
[]     !? _         = Nothing
(x:_)  !? 0         = Just x
(_:xs) !? n         = xs !? (n-1)

myList :: [Int]
myList = [1..9999]
```

[1] http://hackage.haskell.org/package/criterion
[2] http://www.serpentine.com/criterion/tutorial.html
[3] https://www.haskell.org/cabal/users-guide/

```
main :: IO ()
main = defaultMain
  [ bench "index list 9999"
    $ whnf (myList !!) 9998
  , bench "index list maybe index 9999"
    $ whnf (myList !?) 9998
  ]
```

Our version of `!!` shouldn't have anything too surprising going on. We have declared that it's a left-associating infix operator (`infixl`) with a precedence of 9. We haven't talked much about the associativity or fixity of operators since Chapter 2. This is the same associativity and precedence as the normal `!!` operator in `base`.

`Criterion.Main` is the convenience module to import from `criterion`, if you're running benchmarks in a `Main` module. Usually, you'll have a benchmark stanza in your Cabal file that behaves like an executable. It's also possible to do it as a one-off using Stack:

```
$ stack build criterion
$ stack ghc -- -O2 benchIndex.hs
$ ./benchIndex
```

Here, `main` uses a function from `criterion` called `whnf`. The functions `whnf` and `nf` (also in `criterion`), as you might guess, refer to *weak head normal form* and *normal form*, respectively. Weak head normal form, as we said before, evaluates to the first data constructor. That means that if your outermost data constructor is a `Maybe`, it's only going to evaluate enough to find out if it's a `Nothing` or a `Just`—if there is a `Just a`, it won't count the cost of evaluating the `a` value.

Using `nf` would mean you want to include the cost of fully evaluating the `a` as well as the first data constructor. The key when determining whether you want `whnf` or `nf` is to think about what you're trying to benchmark and if reaching the first data constructor will do all the work you're trying to measure or not. We'll talk more about what the difference is and how to decide which you need in a bit.

In our case, what we want is to compare two things: the weak head normal form evaluation of the original indexing operator and that of our safe version, applied to the same long list. We only need weak head normal form, because `!!` and `!?` don't return a data constructor

until they've done the work already, as we can see by taking a look at the first three cases:

```
_      !? n | n < 0 = Nothing
[]     !? _         = Nothing
(x:_)  !? 0         = Just x
```

These cases aren't reached until you've gone through the list as far as you're going to go. The recursive case below doesn't return a data constructor. Instead, it invokes itself repeatedly until one of the above cases is reached. Evaluating to WHNF cannot and does not pause in a self-invoked recursive case like this:

```
(_:xs) !? n         = xs !? (n-1)
-- self function call,
-- not yet in weak head normal form
```

When evaluated to weak head normal form, the above will continue until it reaches the index, you reach the element, or you hit the end of the list. Let us consider an example:

```
[1, 2, 3] !? 2
-- matches final case

(_: [2, 3]) !? 2
  = [2, 3] !? (2-1)
-- not a data constructor, keep going

[2, 3] !? 1
-- matches final case

(_: [3]) !? 1
  = [3] !? (1-1)
-- not a data constructor, keep going

[3] !? 0
-- matches Just case

(x:[]) !? 0 = Just x
-- We stop at Just
```

In the above, we happen to know that `x` is 3, but it'll get thunked if it wasn't opportunistically evaluated on construction of the list.

Next, let's look at the types of the following functions:

```
defaultMain :: [Benchmark] -> IO ()

whnf :: (a -> b) -> a -> Benchmarkable
nf :: Control.DeepSeq.NFData b =>
      (a -> b) -> a -> Benchmarkable
```

The reason it wants a function it can apply an argument to is so that the result isn't shared, which we discussed in the previous chapter. We want it to re-perform the work for each sampling in the benchmark results, so this design prevents that sharing. Keep in mind that if you want to use your own datatype with `nf`, which has an `NFData` constraint, you will need to provide your own instance. You can find examples in the `deepseq` library on Hackage.

Our goal with this example is to equalize the performance difference between `!?` and `!!`. In this case, we've derived the implementation of `!?` from the Haskell Report version of `!!`. Here's how it looks in `base`:

```
-- Go to the Data.List docs in base,
-- and click the source link for (!!).

#ifdef USE_REPORT_PRELUDE
xs !! n | n < 0 =
  error "Prelude.!!: negative index"
[] !! _         =
  error "Prelude.!!: index too large"
(x:_) !! 0      = x
(_:xs) !! n     = xs !! (n-1)
#else
```

However, after you run the benchmarks, you'll find our version based on the above isn't quite as fast.[4] Fair enough! It turns out

[4]Note that if you get weird benchmark results, you'll want to resort to the old programmer's trick of wiping your build. With Stack, you'd run `stack clean`, with Cabal it'd be `cabal clean`. Inexplicable things happen sometimes. You shouldn't need to do this regularly, though.

that most of the time when there's a Report version as well as a non-Report version of a function in `base`, it's because they found a way to optimize it and make it faster. If we look down from the `#else`, we can find the version that replaced it:

```
-- negIndex and tooLarge are a bottom
-- and a const bottom respectively.

{-# INLINABLE (!!) #-}
xs !! n
  | n < 0     = negIndex
  | otherwise =

    foldr
    (\x r k -> case k of
                0 -> x
                _ -> r (k-1))
    tooLarge xs n
```

The non-Report version is written in terms of `foldr`, which often benefits from the various rewrite rules and optimizations attached to `foldr`—rules we will not be explaining here at all, sorry. This version also has a pragma letting GHC know it's OK to inline the code of the function where it's used when the cost estimator thinks it's worthwhile to do so. So, let's change our version of the operator to match this version, in order to make use of those same optimizations:

```
infixl 9 !?
{-# INLINABLE (!?) #-}

xs !? n
  | n < 0     = Nothing
  | otherwise =

      foldr
      (\x r k ->
        case k of
          0 -> Just x
          _ -> r (k-1))
      (const Nothing) xs n
```

If you run this, you'll find that... things have not improved. So, what can we do to improve the performance of our operator?

Well, unless you added one already, you'll notice the type signature is missing. If you add a declaration that the number argument is an `Int`, it should now perform the same as the original:

```
infixl 9 !?
{-# INLINABLE (!?) #-}

(!?) :: [a] -> Int -> Maybe a
xs !? n
  | n < 0     = Nothing
  | otherwise =

      foldr
      (\x r k ->
        case k of
          0 -> Just x
          _ -> r (k-1))
      (const Nothing) xs n
```

Change the function in your module to reflect this, and run the benchmark again to check.

The issue is that the version with an inferred type was defaulting the `Num a => a` to `Integer`, which compiles to a less efficient version of this code than does one that specifies the type `Int`. The `Int` version will turn into a more primitive, faster loop. You can verify this for yourself by specifying the type `Integer` and re-running the code or comparing the GHC Core output for each version.

More on whnf and nf

Let's return now to the question of when we should use `whnf` or `nf`. You want to use `whnf` when the first data constructor is a meaningful indicator of whether the work you're interested in has been done. Consider the simplistic example of a program that is meant to locate some data in a database, say, a person's name and whether there are any known addresses for that person. If it finds any data, it might write that information to a file.

The part you're probably trying to judge the performance of is the lookup function that finds the data and assesses whether it exists, not how fast your computer can write the list of addresses to a file. In that case, what you care about is at the level of weak head normal form, and `whnf` will tell you more precisely how long it is taking to find the data and decide whether you have a `Nothing` or a `Just a`.

On the other hand, if you are interested in measuring the time it takes to write your results, in addition to looking up the data, then you may want to evaluate to normal form. There are times when measuring that makes sense. We'll see some examples shortly.

For now, let us consider each indexing operator, the `!!` that exists in `base` and the one we've written that uses `Maybe` instead of bottoms.

In the former case, the final result has the type `a`. The function doesn't stop recursing until it either returns bottom or the value at that index. In either case, it's done all the work you'd care to measure—traversing the list. Evaluation to WHNF means stopping at your `a` value.

In the latter case with `Maybe`, evaluation to WHNF means stopping at either `Just` or `Nothing`. It won't evaluate the contents of the `Just` data constructor under `whnf`, but it will under `nf`. Either is sufficient for the purposes of the benchmark as, again, we're measuring how quickly this code reaches the value at an index in the list.

Let us consider an example with a few changes:

```
module Main where

import Criterion.Main
import Debug.Trace

myList :: [Int]
myList = trace "myList was evaluated"
         ([1..9999] ++ [undefined])

-- your version of !? here
```

```
main :: IO ()
main = defaultMain
  [ bench "index list 9999"
    $ whnf (myList !!) 9998
  , bench "index list maybe index 9999"
    $ nf (myList !?) 9999
  ]
```

Notice what we did here. We add an `undefined` in what will be the index position 9999. With the `!!` operator, we are accessing the index just before that bottom value, because there is no outer data constructor (such as `Nothing` or `Just`) where we could stop the evaluation. Both `whnf` and `nf` will necessarily force that bottom value.

We also modify the `whnf` to `nf` for the benchmark of `!?`. Now it will evaluate the `undefined` it finds at that index under the bottom in the first run of the benchmark and fail:

```
benchmarking index list maybe index 9999
criterion1: Prelude.undefined
```

A function value that returns bottom instead of a data constructor would have also acted as a stopping point for WHNF. Consider the following:

```
Prelude> (Just undefined) `seq` 1
1
Prelude> (\_ -> undefined) `seq` 1
1
Prelude> ((\_ -> Just undefined) 0) `seq` 1
1
Prelude> ((\_ -> undefined) 0) `seq` 1
*** Exception: Prelude.undefined
```

Much of the time, `whnf` is going to cover the thing you're trying to benchmark.

Making the case for nf

Let us now look at an example of when `whnf` isn't sufficient for benchmarking, something that uses guarded recursion, unlike `!!`:

```
module Main where

import Criterion.Main

myList :: [Int]
myList = [1..9999]

main :: IO ()
main = defaultMain
  [ bench "map list 9999" $
    whnf (map (+1)) myList
  ]
```

The above is an example of *guarded recursion*, because a data constructor is interposed between each recursion step. The data constructor is the cons cell when we're talking about map. Guarded recursion lets us consume the recursion steps up to weak head normal form incrementally, on demand.

Importantly, `foldr` can be used to implement guarded and unguarded recursion, depending *entirely* on what the folding function does rather than any special provision made by `foldr` itself. So what happens when we benchmark this?

```
Linking bin/bench ...
time
  8.844 ns   (8.670 ns .. 9.033 ns)
  0.998 R²   (0.997 R² .. 1.000 R²)
mean
  8.647 ns   (8.567 ns .. 8.751 ns)
std dev
  293.3 ps   (214.7 ps .. 412.9 ps)
variance introduced by outliers:
  57
```

Well, that's suspect. Does it really take 8.8 nanoseconds to traverse a 10,000 element linked list in Haskell? We saw an example of how long it should take, roughly. This is an example of our benchmark being too lazy. The issue is that `map` uses guarded recursion, and the cons cells of the list are interposed between each recursion of `map`. You may recall this from the chapters on lists and folds. So, it ends up evaluating only this far:

```
(_ : _)
```

Ah, that first data constructor. It has neither done the work of incrementing the value nor has it traversed the rest of the list. It's just sitting there at the first cons cell. Using bottoms, you can progressively prove to yourself what `whnf` is evaluating by replacing things and re-running the benchmark:

```
-- is it applying (+1)?
myList = (undefined : [2..9999])

-- Is it going any further in the list?
myList :: [Int]
myList = (undefined : undefined)

-- This should s'plode, because
-- it'll be looking for that first
-- data constructor or -> to stop at.
myList :: [Int]
myList = undefined
```

No matter, we can fix this!
Change this bit:

```
whnf (map (+1)) myList
```

Into this:

```
nf (map (+1)) myList
```

Then we get:

```
time
  122.5 µs   (121.7 µs .. 123.9 µs)
  0.999 R²   (0.998 R² .. 1.000 R²)
mean
  123.0 µs   (122.0 µs .. 125.6 µs)
std dev
  5.404 µs   (2.806 µs .. 9.323 µs)
```

That is considerably more realistic considering we've evaluated the construction of a whole new list. This is slower than the indexing operation, because we're not just kicking a new value out, we're also constructing a new list.

In general, when deciding between `whnf` and `nf`, ask yourself, "When I have reached the first data constructor, have I done most or all of the work that matters?" Be careful not to use `nf` too much. If you have a function that returns a nontrivial data structure or collection for which it's already done all the work to produce, `nf` will make your code look excessively slow and lead you on a wild goose chase.

28.3 Profiling your programs

We're going to do our best to convey what you should know about profiling programs with GHC and what we think is conceptually less well covered, but we aren't going to presume to replace the GHC User Guide. We strongly recommend you read the guide[5] for more information.

Profiling time usage

Sometimes, rather than seeing *how* fast our programs are, we want to know *why* they're slow or fast and where they're spending their time. To that end, we use profiling. First, let's put together a simple example for motivating this:

```
-- profilingTime.hs
module Main where

f :: IO ()
f = do
  print ([1..] !! 999999)
  putStrLn "f"

g :: IO ()
g = do
  print ([1..] !! 9999999)
  putStrLn "g"
```

[5] https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/profiling.html

```
main :: IO ()
main = do
  f
  g
```

Given that we traverse 10 times as much list structure in the case of `g`, we believe we should see something like 10 times as much CPU time spent in `g`. We can do the following to determine if that's the case:

```
$ stack ghc -- -prof -fprof-auto \
> -rtsopts -O2 profile.hs
./profile +RTS -P
cat profile.prof
```

Breaking down what each flag does:

1. `-prof` enables profiling. Profiling isn't enabled by default, because it can lead to slower programs, but this generally isn't an issue when you're investigating performance. Used alone, `-prof` will require you to annotate cost centers manually, places for GHC to mark for keeping track of how much time is spent evaluating something.

2. `-fprof-auto` assigns to all bindings not marked inline a cost center named after the binding. This is fine for little or otherwise not terribly performance-sensitive stuff, but if you're dealing with a large program or one sensitive to perturbations from profiling, it may be better not to use this and instead assign your "SCCs" manually. SCC is what the GHC documentation calls a cost center.

3. `-rtsopts` enables you to pass GHC RTS options to the generated binary. This is optional, so you can get a smaller binary if desired. We need this to tell our program to dump the profile to the `.prof` file named after our program.

4. `-O2` enables the highest level of program optimizations. This is wise if you care about performance, but `-O` by itself also enables optimizations, albeit somewhat less aggressive ones. Either option can make sense when benchmarking; it's a case by case

thing, but most Haskell programmers feel pretty free to default to `-O2`.

After examining the `.prof` file that contains the profiler output, this is roughly what we'll see:

```
Sun Feb 14 21:34 2016
Time and Allocation
  Profiling Report  (Final)

  profile +RTS -P -RTS

total time  =        0.22 secs
  (217 ticks @ 1000 us, 1 processor)
total alloc = 792,056,016 bytes
  (excludes profiling overheads)

COST CENTRE MODULE %time %alloc ticks bytes

g      Main     91.2   90.9 198   720004344
f      Main      8.8    9.1 19    72012568

...more noise snipped...
```

And indeed, 91.2% of the time spent in `g`, 8.8% of the time spent in `f` would seem to validate our hypothesis here.

Time isn't the only thing we can profile. We'd also like to know about the space (or memory) different parts of our program are responsible for using.

Profiling heap usage

We have measured time; now we shall measure space. Well, memory anyway—we're not astrophysicists. We're going to keep this quick and boring so we can get quickly to the good stuff:

```
module Main where

import Control.Monad
```

```
blah :: [Integer]
blah = [1..1000]

main :: IO ()
main =
  replicateM_ 10000 (print blah)
```

```
stack ghc -prof -fprof-auto -rtsopts -O2 loci.hs
./loci +RTS -hc -p
hp2ps loci.hp
```

If you open the `loci.ps` postscript file with your PDF reader of choice, you'll see how much memory the program uses over the time the program runs. Note that you'll need the program to run a minimum amount of time for the profiler to get any samples of the heap size.

28.4 Constant applicative forms

When we're talking about memory usage and sharing in Haskell, we also have to talk about CAFs: constant applicative forms. CAFs are expressions that have no free variables and are held in memory to be shared with all other expressions in a module. They can be literal values or partially applied functions that have no named arguments.

We're going to construct a very large CAF here. Notice we are mapping over an infinite list and want to know how much memory this uses. You might consider betting on *a lot*:

```
module Main where

incdInts :: [Integer]
incdInts = map (+1) [1..]

main :: IO ()
main = do
  print (incdInts !! 1000)
  print (incdInts !! 9001)
  print (incdInts !! 90010)
  print (incdInts !! 9001000)
  print (incdInts !! 9501000)
  print (incdInts !! 9901000)
```

Now, we can profile this:

```
Thu Jan 21 23:25 2016
Time and Allocation
  Profiling Report  (Final)

cafSaturation +RTS -p -RTS
total time  =        0.28 secs
  (283 ticks @ 1000 us, 1 processor)
total alloc = 1,440,216,712 bytes
  (excludes profiling overheads)

COST CENTRE MODULE  %time %alloc
incdInts    Main     90.1  100.0
main        Main      9.9    0.0

...some irrelevant bits elided...

COST CENTRE MODULE no. entries %time %alloc
MAIN        MAIN   45  0       0.0    0.0
 CAF        Main   89  0       0.0    0.0
  incdInts  Main   91  1       90.1  100.0
  main      Main   90  1       9.9    0.0
```

Note how `incdInts` is its own constant applicative form (CAF) here, apart from `main`. And notice the size of the memory allocation. It's because that mapping over an infinite list is a top-level value that can be shared throughout a module, so it must be evaluated and the results held in memory in order to be shared.

CAFs include:

- Values.
- Partially applied functions without named arguments.
- Fully applied functions such as `incdInts` above, although that would be rare in real code.

CAFs can make some programs faster, since you don't have to keep re-evaluating shared values; however, CAFs can become memory-intensive quickly, too. The important takeaway is that, if you find

your program using much more memory than you expected, find the golden CAF and kill it.

Fortunately, CAFs mostly occur in toy code. Real world code is usually pulling the data from somewhere, which avoids the problem of holding large amounts of data in memory.

Let's look at a way to avoid creating a CAF by introducing an argument into our `incdInts` example:

```
module Main where

-- not a CAF
incdInts :: [Integer] -> [Integer]
incdInts x = map (+1) x

main :: IO ()
main = do
  print (incdInts [1..] !! 1000)
```

If you examine the profile:

```
CAF
  main
    incdInts
```

Point-free, top-level declarations will be CAFs, but pointful ones will not. We discussed this to some degree in the previous chapter, as well. The reason the difference matters is often not because of the total allocation reported by the profile, which can be misleading, anyway. Rather, it's important, because lists are as much control structures as data structures, and it's *very* cheap in GHC to construct a list that is thrown away immediately. Doing so might increase how much memory you allocate *in total*, but unlike a CAF, it won't stay in your heap, which may lead to lower peak memory usage and the runtime spending less time collecting garbage.

Indeed, when we run the code above, we do not end up with a standalone CAF. But what if we eta reduce `incdInts`, so that it is a point-free function, instead? In that case, we *will* end up with a CAF. Let's try it out and see what happens with the following, modified code:

```
module Main where

-- Now we get a CAF
incdInts :: [Integer] -> [Integer]
incdInts = map (+1)

main :: IO ()
main = do
  print (incdInts [1..] !! 1000)
```

This time, when you look at the profile, it'll be its own CAF:

```
CAF
  incdInts
  main
    incdInts
```

Great Scott!

It doesn't really change the performance for something so trivial, but you get the idea. The big difference between the two is in the heap profiles. Check them, and you will likely see what we mean.

28.5 Map

We're going to start our exploration of data structures with `Map`. It's worth pointing out here that most of the structures we'll look at are, in some sense, replacements for the lists we have depended on throughout the book. Lists and strings are useful for a lot of things, but they're not the most performant or even the most useful types for structuring your data. What is the most performant or useful for any given program can vary, so we can't give a blanket recommendation that you should always use any one of the structures we're going to talk about. You have to judge that based on what problems you're trying to solve, and use benchmarking and profiling tools to help you fine tune the performance.

Most of the data structures we'll be looking at are in the `containers`[6] library. If you build it, `Map` will come. And also `Sequence` and `Set` and

[6] `http://hackage.haskell.org/package/containers`

some other goodies. You'll notice a lot of the data structures have a similar API, but each are designed for different sets of circumstances.

We've used the `Map` type before to represent associations of unique pairings of keys to values. You may, in particular, remember it from Chapter 14, on testing, where we used it to look up Morse code translations of alphabetic characters. Those were fun times, so carefree and innocent.

The structure of the `Map` type looks like this:

```
data Map k a
  = Bin
    {-# UNPACK #-}
    !Size !k a
    !(Map k a) !(Map k a)
  | Tip

type Size = Int
```

You may recognize the exclamation marks denoting strictness from the sections on bang patterns in the previous chapter. `Tip` is a data constructor for capping off the branch of a tree.

If you'd like to find out about the unpacking of strict fields, which is what the `UNPACK` pragma is for, see the GHC documentation for more information.

What's something that's faster with Map?

Well, lookups by key, in particular, are what it's used for. Consider the following comparison of an association list and `Data.Map`:

```
module Main where

import Criterion.Main
import qualified Data.Map as M

genList :: Int -> [(String, Int)]
genList n = go n []
  where go 0 xs = ("0", 0) : xs
        go n' xs =
          go (n' - 1) ((show n', n') : xs)
```

```
pairList :: [(String, Int)]
pairList = genList 9001

testMap :: M.Map String Int
testMap = M.fromList pairList

main :: IO ()
main = defaultMain
  [ bench "lookup one thing, list" $
    whnf (lookup "doesntExist") pairList
  , bench "lookup one thing, map" $
    whnf (M.lookup "doesntExist") testMap
  ]
```

Association lists such as `pairList` are fine if you need something cheap and cheerful for a very small series of pairs, but you're better off using `Map` by default when you have keys and values. You get a handy set of baked-in functions for looking things up and an efficient means of doing so. Insert operations also benefit from being able to find the existing key-value pair in `Map` more quickly than in association lists.

What's slower with Map?

Using an `Int` as your key type is usually a sign you'd be better off with a `HashMap`, `IntMap`, or `Vector`, depending on the semantics of your problem. If you need good memory density and locality—which will make aggregating and reading values of a large `Vector` faster—then `Map` *could* be inappropriate, and you may want to reach for `Vector`, instead.

28.6 Set

This is also in `containers`. It's like a `Map` but without the "value" part of the key-value pair. You may be asking yourself, why would I only want keys?

When we use `Map`, it has an `Ord` constraint on the functions to ensure that the keys are in order. That is one of the things that makes lookups in `Map` particularly efficient. Knowing that the keys will be

ordered divides the problem space up by halves: if we're looking for the key 6 in a set of keys from 1–10, we don't have to search in the first half of the set, because those numbers are less than 6. `Set`, like `Map`, is structured associatively, not linearly.

Functions with `Set` have the same `Ord` constraint, but now we don't have key-value pairs—we only have keys. Another way to think of it is that the keys are now the values. This means that `Set` represents a unique, ordered set of values.

Here is the datatype for `Set`:

```
data Set a
  = Bin
    {-# UNPACK #-}
    !Size !a !(Set a) !(Set a)
  | Tip

type Size = Int
```

It's effectively equivalent to a `Map` type with unit values:

```
module Main where

import Criterion.Main
import qualified Data.Map as M
import qualified Data.Set as S

bumpIt (i, v) = (i + 1, v + 1)

m :: M.Map Int Int
m = M.fromList $ take 10000 stream
  where stream = iterate bumpIt (0, 0)

s :: S.Set Int
s = S.fromList $ take 10000 stream
  where stream = iterate (+1) 0

membersMap :: Int -> Bool
membersMap i = M.member i m

membersSet :: Int -> Bool
membersSet i = S.member i s
```

```
main :: IO ()
main = defaultMain
  [ bench "member check map" $
    whnf membersMap 9999
  , bench "member check set" $
    whnf membersSet 9999
  ]
```

If you benchmark the program above, you should get very similar results for both structures, with `Map` perhaps being a smidgen slower than `Set`. They're similar, because they're nearly identical data structures in the `containers` library.

There's not much more to say. It has the same pros and cons as `Map`, with the same performance of the core operations, save that you're more limited.

Exercise: Benchmark practice

Make a benchmark to prove for yourself whether `Map` and `Set` have similar performance. Try operations other than membership testing, such as insertion or unions.

28.7 Sequence

`Sequence` is a nifty datatype built atop a data structure called a finger tree. We aren't going to address finger trees in this book, but we encourage you to check them out.[7] `Sequence` appends cheaply on the front and on the back, which avoids a common problem with lists where you can only "cons" cheaply to the front.

Here is the datatype for `Sequence`:

```
newtype Seq a = Seq (FingerTree (Elem a))
```

The name `Elem` is used so that elements and nodes can be distinguished in the types of the implementation. Don't sweat it:

```
newtype Elem a = Elem { getElem :: a }
```

[7] See, for example, *Finger Trees: A Simple General-purpose Data Structure* by Ralf Hinze and Ross Paterson: `http://www.staff.city.ac.uk/~ross/papers/FingerTree.html`.

```
data FingerTree a
    = Empty
    | Single a
    | Deep {-# UNPACK #-} !Int !(Digit a)
           (FingerTree (Node a)) !(Digit a)
```

What's faster with Sequence?

Updates (cons and append) to both ends of the data structure and concatenation are what `Sequence` is particularly known for. You won't want to resort to using `Sequence` by default though, as the list type is often competitive. `Sequence` will have more efficient access to the tail than a list will, however. Here's an example where `Sequence` does better, because the list is a bit big:

```
module Main where

import Criterion.Main
import qualified Data.Sequence as S

lists :: [[Int]]
lists = replicate 10 [1..100000]

seqs :: [S.Seq Int]
seqs =
  replicate 10 (S.fromList [1..100000])

main :: IO ()
main = defaultMain
    [ bench "concatenate lists" $
      nf mconcat lists
    , bench "concatenate sequences" $
      nf mconcat seqs
    ]
```

Note that in the example above, a substantial amount of the time in the benchmark is spent traversing the data structure, because we're evaluating to normal form to ensure all the work is done. Incidentally, this accentuates the difference between a list and `Sequence`, because it's faster to index or traverse a sequence than a list. Consider the following:

```
module Main where

import Criterion.Main
import qualified Data.Sequence as S

lists :: [Int]
lists = [1..100000]

seqs :: S.Seq Int
seqs = S.fromList [1..100000]

main :: IO ()
main = defaultMain
    [ bench "indexing list" $
      whnf (\xs -> xs !! 9001) lists
    , bench "indexing sequence" $
      whnf (flip S.index 9001) seqs
    ]
```

The difference between list and `Sequence` in the above when we run the benchmark is two *orders of magnitude*.

What's slower with Sequence?

`Sequence` is a persistent data structure like `Map`, so the memory density isn't as good as it is with `Vector` (we're getting there). Indexing by `Int` will be faster with `Vector`, as well. List will be faster with consing and concatenation in some cases, if the lists are small. When you know you need cheap appending to the end of a long list, it's worth giving `Sequence` a try, but it's better to base the final decision on benchmarking data, if performance matters.

28.8 Vector

The next data structure we're going to look at is not in `containers`. It's in its own library called, unsurprisingly, `vector`[8]. You'll notice it says vectors are "efficient arrays." We're not going to look at arrays, or Haskell's `Array` type, in this book, though you may already be familiar with the idea.

[8] `https://hackage.haskell.org/package/vector`

One rarely uses arrays, or more specifically, `Array`[9] in Haskell. `Vector` is almost always what you want instead of an array. The default `Vector` type is implemented as a slice wrapper of `Array`; we can see this in the definition of the datatype:

```
-- | Boxed vectors, supporting
--   efficient slicing.
data Vector a =
     Vector {-# UNPACK #-} !Int
            {-# UNPACK #-} !Int
            {-# UNPACK #-} !(Array a)
     deriving ( Typeable )
```

There are many variants of `Vector`, although we won't discuss them all here. These include boxed, unboxed, immutable, mutable, and storable vectors, but the plain version above is the usual one you'd use. We'll talk about mutable vectors in their own section. *Boxed* means the vector can reference any datatype you want; *unboxed* represents raw values without pointer indirection.[10] The latter can save a lot of memory but is limited to types like `Bool`, `Char`, `Double`, `Float`, `Int`, `Word`, and tuples of unboxable values. Since a newtype is unlifted and doesn't introduce any pointer indirection, you can unbox any newtype that contains an unboxable type.

When does one want a Vector in Haskell?

You want a `Vector` when:

- You need memory efficiency close to the theoretical maximum for the data you are working with.
- Your data access is almost exclusively in terms of indexing via an `Int` value.
- You want uniform performance for accessing each element in the data structure.

[9] `http://hackage.haskell.org/package/array`

[10] This book isn't the right place to talk about what pointers are in detail. Briefly, they are references to things in memory, rather than the things themselves.

- You will construct a `Vector` once and read it many times (alternatively, you plan to use a mutable `Vector` for ongoing, efficient updates).

What's this about slicing?

Remember this from the definition of `Vector`?

```
-- | Boxed vectors, supporting
--   efficient slicing.
```

Slicing refers to slicing off a portion, or creating a sub-array. The `Vector` type is designed for making slicing much cheaper than it otherwise would be. Consider the following:

```
module Main where

import Criterion.Main
import qualified Data.Vector as V

slice :: Int -> Int -> [a] -> [a]
slice from len xs =
  take len (drop from xs)

l :: [Int]
l = [1..1000]

v :: V.Vector Int
v = V.fromList [1..1000]

main :: IO ()
main = defaultMain
    [ bench "slicing list" $
      whnf (head . slice 100 900) l
    , bench "slicing vector" $
      whnf (V.head . V.slice 100 900) v
    ]
```

If you run this benchmark, `Vector` should be faster than list. Why? Because when we construct that new `Vector` with `V.slice`, all it has to do is the following:

```
-- from Data.Vector

instance G.Vector Vector a where
  -- other methods elided
  {-# INLINE basicUnsafeSlice #-}
  basicUnsafeSlice j n (Vector i _ arr) =
    Vector (i+j) n arr
```

What makes `Vector` nicer than lists and `Array` in this respect is that when you construct a slice or view of another `Vector`, it doesn't have to construct as much new data. It returns a new wrapper around the original underlying array with a new index and offset with a reference to the same original `Array`. Doing the same with an ordinary `Array` or a list would have required copying more data. Speed comes from being sneaky and skipping unnecessary work.

Updating vectors

Persistent vectors are not great at handling updates on an ongoing basis, but there are some situations in which they can surprise you. One such case is fusion.[11] Fusion, or loop fusion, means that as an optimization, the compiler can fuse several loops into one megaloop and do the whole operation in one pass:

```
module Main where

import Criterion.Main
import qualified Data.Vector as V

testV' :: Int -> V.Vector Int
testV' n =
  V.map (+n) $ V.map (+n) $
    V.map (+n) $ V.map (+n)
    (V.fromList [1..10000])
```

[11] See Duncan Coutts, *Stream Fusion*, for more on this topic: `http://fun.cs.tufts.edu/stream-fusion.pdf`.

```
testV :: Int -> V.Vector Int
testV n =
  V.map ( (+n) . (+n)
        . (+n) . (+n) )
        (V.fromList [1..10000])

main :: IO ()
main = defaultMain
  [ bench "vector map pre-fused" $
      whnf testV 9998
  , bench "vector map will be fused" $
      whnf testV' 9998
  ]
```

The `vector` library has loop fusion built in, so in many cases, such as with mapping, you won't construct four vectors just because you map four times. Through the use of GHC rewrite rules[12], the code in `testV'` above will change into what you see in `testV`. It's worth noting that part of the reason this optimization is sound is because we know what code performs effects and what does not, because we have types.

However, loop fusion isn't a panacea, and there will be situations in which you will want to change one element at a time, selectively. Let's consider more efficient ways of updating vectors. We're going to use the `//` operator from the `vector` library here. It's a batch update operator that allows you to modify several elements of a vector at once:

```
module Main where

import Criterion.Main
import Data.Vector ((//))
import qualified Data.Vector as V

vec :: V.Vector Int
vec = V.fromList [1..10000]
```

[12] https://wiki.haskell.org/GHC/Using_rules

```
slow :: Int -> V.Vector Int
slow n = go n vec
  where go 0 v = v
        go n v = go (n-1) (v // [(n, 0)])

batchList :: Int -> V.Vector Int
batchList n = vec // updates
  where updates =
          fmap (\n -> (n, 0)) [0..n]

main :: IO ()
main = defaultMain
  [ bench "slow" $ whnf slow 9998
  , bench "batch list" $
    whnf batchList 9998
  ]
```

The issue with the first example is that we're using a batch API… but not with a batch of updates. It's much cheaper (500–1000x in our tests) to construct the list of updates all at once and then feed them to the `//` function. We can make it even faster still by using the `updates` function, which uses a vector of updates:

```
batchVector :: Int -> V.Vector Int
batchVector n = V.unsafeUpdate vec updates
  where updates =
          fmap (\n -> (n, 0))
          (V.fromList [0..n])
```

Benchmarking this version should get you code that is about 1.4x faster. But we're greedy. So we want more.

Mutable vectors

> "Moria! Moria! Wonder of the Northern world. Too deep we delved there, and woke the nameless fear."
>
> —Glóin, *The Fellowship of the Ring*

What if we want something even faster? Although many people don't realize this about Haskell, we can obtain the benefits of in-place

updates if we so desire. What sets Haskell apart is that we cannot do so in a way that compromises referential transparency or the nice equational properties our expressions have. First, we'll demonstrate how to do this, then we'll talk about how this is designed to behave predictably.

Here comes an example:

```
module Main where

import Control.Monad.Primitive
import Control.Monad.ST
import Criterion.Main

import qualified Data.Vector as V
import qualified Data.Vector.Mutable as MV
import qualified
       Data.Vector.Generic.Mutable as GM

mutableUpdateIO
  :: Int
  -> IO (MV.MVector RealWorld Int)
mutableUpdateIO n = do

  mvec <- GM.new (n+1)
  go n mvec
  where go 0 v = return v
        go n v =
          (MV.write v n 0) >> go (n-1) v

mutableUpdateST :: Int -> V.Vector Int
mutableUpdateST n = runST $ do

  mvec <- GM.new (n+1)
  go n mvec
  where go 0 v = V.freeze v
        go n v =
          (MV.write v n 0) >> go (n-1) v
```

```
main :: IO ()
main = defaultMain
  [ bench "mutable IO vector"
    $ whnfIO (mutableUpdateIO 9998)
  , bench "mutable ST vector"
    $ whnf mutableUpdateST 9998
  ]
```

We're going to talk about `runST` shortly. For the moment, let's concentrate on the fact that the mutable `IO` version is approximately 7000x faster than the original unbatched loop in our tests. The `ST` version is about 1.5x slower than the `IO` version but still very fast. The added time in the `ST` version is from freezing the mutable vector into an ordinary vector. We won't explain `ST` fully here, but, as we'll see, it can be handy when you want to temporarily make something mutable and ensure that no mutable references are exposed outside of the `ST` monad.

Here are the timings we get with the various vector operations on our computer:

Variant	Microseconds
`slow`	133,600
`batchList`	244
`batchVector`	176
`mutableUpdateST`	32
`mutableUpdateIO`	19

Notably, most of the performance improvement is from not doing something silly. Don't resort to the use of mutation via `IO` or `ST` except where you *know* it's necessary. It's worth noting that since our test involves updating almost 10,000 indices in the vector, we spend an average of 1.9 *nanoseconds* per update when we use mutability and 17.6 ns when we do it as a batch with a persistent vector.

A sidebar on the ST Monad

`ST` can be thought of as a mutable variant of the strict `State` monad. From another angle, it could be thought of as `IO` restricted exclusively to mutation, which is guaranteed safe.

Safe how? `ST` is relying on the principle behind that old philosophical saw about the tree that falls in the forest with no one to see it fall. The idea behind this "morally effect-free" understanding of mutable state was introduced in the paper *Lazy Functional State Threads*.[13] It unfreezes your data, mutates it, then refreezes it so that it can't mutate anymore. Thus, it manages to perform a mutation but still maintain referential transparency.

For `ST` to work properly, the code that mutates the data must not get reordered by the optimizer or otherwise monkeyed with, much like the code in `IO`. So there must be something underlying the type that prevents GHC from ruining our day. Let us examine the `ST` type:

```
Prelude> import Control.Monad.ST
Prelude> :info ST
type role ST nominal representational
newtype ST s a =
  GHC.ST.ST (GHC.ST.STRep s a)
...
Prelude> import GHC.ST
Prelude> :info STRep
type STRep s a =
     GHC.Prim.State# s
  -> (# GHC.Prim.State# s, a #)
```

There's a bit of ugliness in here that shouldn't be too surprising now that you've seen GHC Core in the previous chapter. What's important is that `s` is getting its type from the thing you're mutating, but it has no value-level witness. The `State` monad here is therefore erasable; it can encapsulate this mutation process and then melt away.

It's important to understand that `s` isn't the state you're mutating. The mutation is a side effect of having entered the *closures* that perform the effect. This strict, unlifted state transformer monad happens to structure your code in a way that preserves the intended order of the computations and their associated effects.

[13] See John Launchbury and Simon Peyton Jones, *Lazy Functional State Threads*: `https://www.microsoft.com/en-us/research/wp-content/uploads/1994/06/lazy-functional-state-threads.pdf`.

By closures here, we mean lambda expressions. Of course we do, because this whole book is one giant lambda expression under the hood. Entering each lambda performs its effects. The effects of a series of lambdas are not batched, because the ordering is important when we're performing effects, as each new expression might depend on the effects of the previous one. The effects are performed and then, if the value that should result from the computation is also going to be used, the value is evaluated.

So what is the `s` type for? Well, while it's all well and good to ask politely that programmers freeze a mutable reference into a persistent, immutable data structure as the final result… you can't count on that. So `ST` enforces this at compile time by preventing `s` from unifying with anything outside of the `ST Monad`. The trick for this is called existential quantification,[14] but having said this won't necessarily mean anything to you right now. It does prevent you from accidentally leaking mutable references to code outside `ST`, however, which could then lead to code that does unpredictable things, depending on the state of the bits in memory.

Avoid dipping in and out of `ST` over and over. The thaws and freezes will cost you in situations where it might have been better to just use `IO`. Batching your mutations is best for performance and code comprehensibility.

Exercises: Vector

Set up a benchmark harness with `criterion` to profile how much memory boxed and unboxed vectors containing the same data use. You can combine this with a benchmark to give it something to do for a few seconds. We're not giving you a lot to go on here, because you're going to have to learn to read documentation and source code, anyway.

28.9 String types

The title above is a slight bit of a misnomer, as we'll be discussing two string (or text) types and one type for representing sequences of bytes. Here's a brief explanation of `String`, `Text`, and `ByteString`:

[14] See the Haskell Wikibook: *Existentially quantified types*. `https://en.wikibooks.org/wiki/Haskell/Existentially_quantified_types`.

String

You know what `String` is. It's a type alias for a list of `Char`, yet underneath it's not quite as simple as an actual list of `Char`.

One of the benefits of using strings is their simplicity, and they're easy enough to explain. For most demonstration and toy program purposes, they're fine.

However, like lists themselves, they can be infinite. The memory usage for very large strings can get out of control rapidly. Furthermore, character-by-character indexing into a `String` is inefficient—the time taken to do a lookup increases proportionately with the length of the underlying list.

Text

This one comes in the `text`[15] library on Hackage. `Text` is best for when you have plain text but need to store the data more efficiently, particularly as it concerns memory usage.

We used this one a bit in previous chapters, when we were playing with `OverloadedStrings`. The benefits here are that you get:

- Compact representation in memory.
- Efficient indexing into the string.

However, `Text` is encoded as UTF-16, and that isn't what most people expect, given UTF-8's popularity. In `Text`'s defense, UTF-16 is often faster due to cache friendliness as well as chunkier and more predictable memory accesses. That said, the authors of the `text` library are, at the time of writing, thinking about changing `Text`'s internal representation to UTF-8, anyway. In the meantime, if you really need UTF-8 now, you can check out the related libraries, `text-utf8` and `text-short`.

Don't trust your gut, measure

We mentioned above that `Text` has a more compact memory representation than the usual Haskell `String` type, but what do you think the memory profile for the following program will look like? High first, then low or low first, then high?

[15] `http://hackage.haskell.org/package/text`

```
module Main where

import Control.Monad
import qualified Data.Text as T
import qualified Data.Text.IO as TIO
import qualified System.IO as SIO

-- replace "/usr/share/dict/words"
-- as appropriate
dictWords :: IO String
dictWords =
  SIO.readFile "/usr/share/dict/words"

dictWordsT :: IO T.Text
dictWordsT =
  TIO.readFile "/usr/share/dict/words"

main :: IO ()
main = do
  replicateM_ 1000 (dictWords >>= print)
  replicateM_ 1000
    (dictWordsT >>= TIO.putStrLn)
```

The issue is that `Text` goes ahead and loads the entire file into memory each of the ten times you force the IO action. The `readFile` operation for `String`, however, is lazy and only reads as much of the file into memory as needed to print the data to `stdout`. The proper way to handle incrementally processing data is with streaming,[16] but this is not something we'll cover in detail in this book. However, there is a *lazy* way we could change this.

Add the following code to the module you already made for profiling `String` and `Text`:

```
import qualified Data.Text.Lazy as TL
import qualified Data.Text.Lazy.IO as TLIO

dictWordsTL :: IO TL.Text
dictWordsTL =
  TLIO.readFile "/usr/share/dict/words"
```

[16] For example, with the `pipes` library: `https://wiki.haskell.org/Pipes`.

```
main :: IO ()
main = do
  replicateM_ 1000 (dictWords >>= print)
  replicateM_ 1000
    (dictWordsT >>= TIO.putStrLn)
  replicateM_ 1000
    (dictWordsTL >>= TLIO.putStrLn)
```

Now, you should see memory usage plummet again after a middle plateau, because you're reading in as much text as necessary to print, and you're able to deallocate as you go. This allows us to see some, but not all, of the benefits of streaming, and we do strongly recommend using streaming rather than relying on a lazy `IO` API.

ByteString

This is not a string. Or text. Not necessarily, anyway. A `ByteString` is a sequence of bytes represented (indirectly) as a vector of `Word8` values. Text on a computer is made up of bytes, but it has to have a particular encoding in order for it to be text. Encodings of text can be ASCII, UTF-8, UTF-16, or UTF-32—usually UTF-8 or UTF-16. The `Text` type encodes the data as UTF-16, partly for performance. It's often faster to read larger chunks of data at a time from memory, so the 16-bits per code point encoding of UTF-16 performs better in most cases.

The main benefit of `ByteString` is that it's easy to use via the `OverloadedStrings` extension, although it's bytes instead of text. This addresses a larger problem space than mere text.

The flip side of that, of course, is that it encompasses byte data that isn't comprehensible text. That's a drawback if you didn't mean to permit non-text byte sequences in your data.

ByteString examples

Here's an example highlighting that `ByteStrings` are not always text:

```
{-# LANGUAGE OverloadedStrings #-}

module BS where
```

```
import qualified Data.Text.IO as TIO
import qualified Data.Text.Encoding as TE
import qualified Data.ByteString.Lazy as BL
-- https://hackage.haskell.org/package/zlib
import qualified
  Codec.Compression.GZip as GZip
```

We're going to use the `gzip` compression format to compress some data. This is so we have an example of data that includes bytes that aren't a valid text encoding:

```
input :: BL.ByteString
input = "123"

compressed :: BL.ByteString
compressed = GZip.compress input
```

The `GZip` module expects a lazy `ByteString`, probably so that it's streaming friendly:

```
main :: IO ()
main = do

  TIO.putStrLn $ TE.decodeUtf8 (s input)
  TIO.putStrLn $
    TE.decodeUtf8 (s compressed)
  where s = BL.toStrict
```

The encoding module in the `text` library expects strict `ByteString` values, so we have to make them strict before attempting a decoding. The second text decode above will fail, because there will be a byte that isn't recognizably correct as an encoding of text information.

ByteString traps

You might think to yourself at some point, "I'd like to convert a `String` to a `ByteString`!" This is a perfectly reasonable thing to want to do, but many Haskellers will mistakenly use the `Char8` module in the `bytestring` library when that is not really what they want. The `Char8` module is just a convenience for data that mingles bytes and ASCII

data.[17] It doesn't work for Unicode and shouldn't be used anywhere there's even a *hint* of a possibility that there could be Unicode data. For example:

```
module Char8ProllyNotWhatYouWant where

import qualified Data.Text as T
import qualified Data.Text.Encoding as TE
import qualified Data.ByteString as B

import qualified
  Data.ByteString.Char8 as B8
-- utf8-string
import qualified
  Data.ByteString.UTF8 as UTF8

-- Manual Unicode encoding of Japanese text
-- GHC Haskell allows UTF-8 in source files
s :: String
s = "\12371\12435\12395\12385\12399\12289\
\20803\27671\12391\12377\12363\65311"

utf8ThenPrint :: B.ByteString -> IO ()
utf8ThenPrint =
  putStrLn . T.unpack . TE.decodeUtf8

throwsException :: IO ()
throwsException =
  utf8ThenPrint (B8.pack s)

bytesByWayOfText :: B.ByteString
bytesByWayOfText = TE.encodeUtf8 (T.pack s)

-- letting utf8-string do it for us
libraryDoesTheWork :: B.ByteString
libraryDoesTheWork = UTF8.fromString s
```

[17] Since ASCII is 7 bits and `Char8` is 8 bits, you could use the eighth bit to represent Latin-1 characters. However, since you will usually intend to convert the `Char8` data to encodings like UTF-8 and UTF-16, which use the eighth bit differently, that would be unwise.

```
thisWorks :: IO ()
thisWorks = utf8ThenPrint bytesByWayOfText

alsoWorks :: IO ()
alsoWorks =
  utf8ThenPrint libraryDoesTheWork
```

Then, we go to run the code that attempts to get a `ByteString` via the `Char8` module that contains Unicode:

```
Prelude> throwsException
*** Exception: Cannot decode byte '\x93':
  Data.Text.Internal.Encoding.decodeUtf8:
    Invalid UTF-8 stream
```

You can use `ord` from `Data.Char` to get the `Int` value of the byte of a character:

```
Prelude> import Data.Char (ord)
Prelude> :t ord
ord :: Char -> Int
Prelude> ord 'A'
65
Prelude> ord '\12435'
12435
```

The second example seems obvious, but when the data is represented natively on your computer, this is more useful. Use non-English websites to get sample data to test.

We can now use the ordering of characters to find the first character that breaks `Char8`:

```
Prelude> xs = ['A'..'\12435']
Prelude> cs = map (:[]) xs
Prelude> mapM_ (utf8ThenPrint . B8.pack) cs
...bunch of output...
```

Then, to narrow this down, we know we need to find what comes after the tilde and the `\DEL` character:

```
...some trial and error...
Prelude> f = take 3 $ drop 60
Prelude> mapM_ putStrLn (f cs)
}
~
```

Hmm, OK, but where is this in the ASCII table? We can use `chr`, the opposite of the `ord` function from `Data.Char`, to determine this:

```
Prelude> import Data.Char (chr)
Prelude> :t chr
chr :: Int -> Char
Prelude> map chr [0..128]
...prints the first 129 characters...
```

What it prints corresponds to the ASCII table, which is how UTF-8 represents the same characters. Now, we can use this function to narrow down precisely what our code fails at doing:

This works fine:

```
Prelude> utf8ThenPrint (B8.pack [chr 127])
```

And this fails:

```
Prelude> utf8ThenPrint (B8.pack [chr 128])
*** Exception: Cannot decode byte '\x80':
  Data.Text.Internal.Encoding.decodeUtf8:
    Invalid UTF-8 stream
```

Don't use Unicode characters with the `Char8` module! This problem isn't exclusive to Haskell—all programming languages must acknowledge the existence of different encodings for text.

Char8 is bad mmmmmkay The `Char8` module is *not* for Unicode or for text more generally! The `pack` function it contains is for ASCII data only! This fools programmers, because the UTF-8 encoding of the English alphabet with some Latin extension characters intentionally overlaps exactly with the same bytes ASCII uses to encode those characters. So, the following will work but is wrong in principle:

```
Prelude> utf8ThenPrint (B8.pack "blah")
blah
```

Getting a UTF-8 bytestring via the `text` or `utf8-string` libraries works fine, however, as you'll see if you take a look at the result of `thisWorks` and `alsoWorks`.

When would I use ByteString instead of Text for textual data?

This does happen sometimes, usually because you want to keep data that arrives in a UTF-8 encoding in UTF-8. Often this happens, because you read UTF-8 data from a file or network socket, and you don't want the overhead of bouncing it into and back out of `Text`. If you do this, you might want to use newtypes to avoid accidentally mixing this data with non-UTF-8 bytestrings.

28.10 Chapter exercises

Difference list

Lists are really nice, but they don't append or concatenate cheaply. We covered `Sequence` as one potential solution to this, but there's a simpler data structure that solves slow appending specifically, the *difference list*!

Rather than justify and explain difference lists, part of this exercise is figuring out what it does and why (although feel free to look up the documentation on Hackage). Attempt the exercise before resorting to the tutorial in the follow-up reading. First, the `DList` type is built on top of ordinary lists, but it's a function:

```
newtype DList a = DL { unDL :: [a] -> [a] }
```

The API that follows is based on code by Don Stewart and Sean Leather. Here's what you need to write:

1. ```
   empty :: DList a
   empty = undefined
   {-# INLINE empty #-}
   ```

2. ```
   singleton :: a -> DList a
   singleton = undefined
   {-# INLINE singleton #-}
   ```

3. ```
   toList :: DList a -> [a]
   toList = undefined
   {-# INLINE toList #-}
   ```

4. ```
   -- Prepend a single element to a dlist.
   infixr `cons`
   cons        :: a -> DList a -> DList a
   cons x xs   = DL ((x:) . unDL xs)
   {-# INLINE cons #-}
   ```

5. ```
   -- Append a single element to a dlist.
   infixl `snoc`
   snoc :: DList a -> a -> DList a
   snoc = undefined
   {-# INLINE snoc #-}
   ```

6. ```
   -- Append dlists.
   append :: DList a -> DList a -> DList a
   append = undefined
   {-# INLINE append #-}
   ```

What's so nifty about `DList` is that `cons`, `snoc`, and `append` all take the same amount of time no matter how long the dlist is. That is to say, they take a *constant* amount of time rather than growing with the size of the data structure.

Your goal is to get the following benchmark harness running with the performance expected:

```
schlemiel :: Int -> [Int]
schlemiel i = go i []
  where go 0 xs = xs
        go n xs = go (n-1) ([n] ++ xs)

constructDlist :: Int -> [Int]
constructDlist i = toList $ go i empty
  where go 0 xs = xs
        go n xs =
          go (n-1)
          (singleton n `append` xs)
```

```
main :: IO ()
main = defaultMain
  [ bench "concat list" $
    whnf schlemiel 123456
  , bench "concat dlist" $
    whnf constructDlist 123456
  ]
```

If you run the code above, the `DList` variant should be about twice as fast.

A simple queue

We're going to write another data structure in terms of list, but this time it'll be a queue. The main feature of queues is that we can add elements to the front cheaply and take items off the back of the queue cheaply, as well.

```
-- From Okasaki's Purely
-- Functional Data Structures
data Queue a =
  Queue { enqueue :: [a]
        , dequeue :: [a]
        } deriving (Eq, Show)

-- adds an item
push :: a -> Queue a -> Queue a
push = undefined

pop :: Queue a -> Maybe (a, Queue a)
pop = undefined
```

We're going to give you less code this time, but your task is to implement the data structure above and write a benchmark comparing it against performing alternating pushes and pops from a queue based on a single list—alternating so that you can't take advantage of reversing the list after a long series of pushes in order to perform a long series of pops efficiently.

Don't forget to handle the case where the dequeue is empty and you need to shift items from the enqueue to the dequeue. You need to do so without violating the principle of "first come, first served."

Lastly, benchmark it against `Sequence`. Come up with a variety of tests. Add additional operations for your `Queue` type, if you want.

28.11 Follow-up resources

1. Bryan O'Sullivan. *A criterion tutorial*.
   `http://www.serpentine.com/criterion/tutorial.html`
2. Tom Ellis. *Demystifying DList*.
   `http://h2.jaguarpaw.co.uk/posts/demystifying-dlist`
3. Haskell Wiki. *GHC/Memory Management*.
   `https://wiki.haskell.org/GHC/Memory_Management`
4. Haskell Wiki. *Performance*.
   `https://wiki.haskell.org/Performance`
5. The Glorious Glasgow Haskell Compilation System User's Guide. *Pragmas*. See the section on `UNPACK`.
6. Johan Tibell. *High Performance Haskell*.
   `http://johantibell.com/files/slides.pdf`
7. Johan Tibell. *Haskell Performance Patterns*.
8. Johan Tibell. *Faster persistent data structures through hashing*.
   `http://johantibell.com/files/galois-2011.pdf`
9. John Launchbury and Simon Peyton Jones. *Lazy Functional State Threads*.
10. Don Stewart. *Write Haskell as fast as C: exploiting strictness, laziness and recursion*.
11. Jan Stolarek. *Haskell as fast as C: A case study*.
12. Ian Ross. *Haskell FFT 11: Optimisation Part 1*.
13. Edsko de Vries. *Understanding the RealWorld*.
14. Duncan Coutts. *Stream Fusion*.
    `http://fun.cs.tufts.edu/stream-fusion.pdf`

15. Chris Okasaki. *Purely Functional Data Structures*.
https://www.cs.cmu.edu/~rwh/theses/okasaki.pdf

Chapter 29

IO

> In those days, many successful projects started out as graffitis on a beer mat in a very, very smoky pub.
>
> Peter J. Landin

29.1 IO

You should be proud of yourself for making it this far into the book. You're juggling monads. You've defeated an army of monad transformers. You're comfortable with using algebraic structures in type class form. You've got a basic understanding of how Haskell terms evaluate, non-strictness, and sharing. With those things in hand, let's talk about `IO`.

We've used the `IO` type at various times throughout this book, with only cursory explanation. You no doubt know that we use this type in Haskell as a means of keeping our chocolate separate from our peanut butter—that is, our pure functions from our effectful ones. Perhaps you're wondering how it all works, what's underneath that opaque type. To many people, `IO` seems mysterious.

An effectful function is one that has an observable impact on the environment in which it is evaluated, other than computing and returning a result. Examples of effects includes writing to standard output (like `putStrLn`), reading from standard input (`getChar`), or modifying state destructively (`ST`). Implicit here is that such effects almost always require the code to be evaluated in a particular order. Haskell expressions that aren't in `IO` will always return the same result regardless of what order they are evaluated in; we lose this guarantee and others besides, however, once `IO` is introduced.

Most explanations of the `IO` type in Haskell don't help much, either. They seem designed to confuse the reader rather than convey anything useful. Don't look now, but somebody this very minute is writing an explanation of `IO` that uses van Laarhoven free monads and costate comonad coalgebras to explain something that's much simpler than any of those topics.

We're not going to do that here. We will instead try to demystify `IO` a bit. The important thing about `IO` is that it's a special kind of datatype that disallows sharing in some cases.

In this chapter, we will:

- Explain how `IO` works operationally.
- Explore what it should mean to you when you read a type that has `IO` in it.

- Provide a bit more detail about the `IO` instances of `Functor`, `Applicative`, and `Monad`.

29.2 Where IO explanations go astray

A lot of explanations of `IO` are misleading or muddled. We've already alluded to the overwrought and complex explanations of `IO`. Others call it "the `IO Monad`" and seem to equate `IO` with `Monad`. While `IO` is a type that has a `Monad` instance, it is not *only* a `Monad`, and monads are not only `IO`. Other presentations imply that once you enter `IO`, you destroy purity and referential transparency. And some references don't bother to say much about `IO`, because the fact that it remains opaque won't stop you from doing most of what you want to use it for anyway.

We want to offer some guidance in critically evaluating explanations of `IO`. Let us consider one of the most popular explanations of `IO`, the one that attempts to explain `IO` in terms of `State`.

Burn the State to the ground!

The temptation to use `State` to get someone comfortable with the idea of `IO` is strong. Give the following passage early in the documentation to `GHC.IO` a gander:

> The `IO` Monad is just an instance of the `ST` monad, where the state is the real world.

The motivation for these explanations is easy to understand when you look at the underlying types:

```
-- from ghc-prim
import GHC.Prim
import GHC.Types

newtype State s a
  = State {runState :: s -> (a, s)}

-- :info IO
newtype IO a =
  IO (State# RealWorld
      -> (# State# RealWorld, a #))
```

Yep, it sure looks like `State`! However, this is less meaningful or helpful than you might at first think.

The issue with this explanation is that you don't usefully see or interact with the underlying `State#`[1] in `IO`. It's not `State` in the sense that one uses `State`, `StateT`, or even `ST`, although the behavior of the `s` here is similar to that of `ST`.

The `State` here is a signalling mechanism for telling GHC what order your IO actions are in and what a unique IO action is. If we look through the `GHC.Prim` documentation, we can find the following:

> `State#` is the primitive, unlifted type of states. It has one type parameter, thus `State# RealWorld`, or `State# s`, where `s` is a type variable. The only purpose of the type parameter is to keep different state threads separate. It is represented by nothing at all.
>
> `RealWorld` is deeply magical. It is *primitive*, but it is not *unlifted* (hence `ptrArg`). We never manipulate values of type `RealWorld`; it's only used in the type system, to parameterise `State#`.

When it says that `RealWorld` is "represented by nothing at all," it means it literally uses zero bits of memory. The state tokens underlying the `IO` type are erased during compile time and add no overhead to your runtime. So the problem with explaining `IO` in terms of `State` is not precisely that it's wrong; it's that it's not a `State` you can meaningfully interact with or control in the way you'd expect from the other `State` types.

29.3 The reason we need this type

So, let's try to move from there to an understanding of `IO` that is meaningful to us in our day-to-day Haskelling. `IO` primarily exists to give us a way to order operations and to disable some of the sharing that we talked so much about in Chapter 27, on non-strictness.

GHC is ordinarily free to do a lot of reordering of operations, delaying of evaluation, sharing of named values, duplicating code via

[1] The `#` indicates a *primitive type*. These are types that cannot be defined in Haskell itself and are exported by the `GHC.Prim` module.

inlining, and other optimizations in order to increase performance. The main thing the `IO` type does is turn off most of those abilities.

What?

No, really. That's a lot of it.

Order and chaos

As we've seen in previous chapters, GHC can normally reorder operations. This is disabled in `IO` (as in `ST`). IO actions are instead enclosed within nested lambdas—nesting is the only way to ensure that actions are sequenced within a pure lambda calculus.

Nesting lambdas is how we guarantee that this:

```
main = do
  putStr   "1"
  putStr   "2"
  putStrLn "3"
```

Will output `"123"`, and we want that guarantee. The underlying representation of `IO` allows the actions to be nested and therefore sequenced.

When we enter a lambda expression, any effects that need to be performed will be performed first, before any computations are evaluated. Then, if there is a computation to evaluate, that may be evaluated next, before we enter the next lambda to perform the next effect, and so on. We've seen how this kind of thing plays out already—think of the parsers that perform the effect of moving a "cursor" through a text without reducing to any value. Also, recall what we saw with `ST` and mutable vectors.

In fact, the reason we have `Monad` is because it is a means of abstracting away the nested lambda noise that underlies `IO`.

29.4 Sharing

In addition to enforcing ordering, `IO` turns off a lot of the sharing we also discussed in the chapter on non-strictness. As we'll soon see, it doesn't disable all forms of sharing—it couldn't, because all Haskell

programs have a `main` action with an obligatory `IO` type. But we'll get to that in a moment.

For now, let's turn our attention to what sharing is disabled and why. Usually in Haskell, we're pretty confident that if a function is going to be evaluated at all, it will result in a value of a certain type, bearing in mind that this could be a `Nothing` value or an empty list. When we declare the type, we say, "If this is evaluated at all, we will have a value of this type as a result."

But with the `IO` type, you're not guaranteed anything. Values of type `IO a` are not an `a`; they're a description of how you might get an `a`. Something of type `IO String` is not a computation that, if evaluated, will result in a `String`; it's a description of how you might get that `String` from the "real world," possibly performing effects along the way. Describing `IO` actions does not perform them, just as having a recipe for a cake does not give you a cake.[2]

In this environment, where you do not have a value but only a means of getting a value, it wouldn't make sense to say that value could be shared.

The time has come

So, one of the key features of `IO` is that it turns off sharing. Let's use an example to think about why we want this. We have this library function that gets the current UTC time from the system clock:

```
-- from Data.Time.Clock
getCurrentTime :: IO UTCTime
```

Without `IO` preventing sharing, how would this work? If you fetch the time once, it would share that result, and the time would be whatever time it was the *first time* you forced it. Unfortunately, this is not a means of stopping time; we would continue to age, but your program wouldn't work at all the way you'd intended.

But if that's so, and it's clearly a value with a name that *could* be shared, why isn't it?

```
getCurrentTime :: IO UTCTime
--                ^-- that right there
```

[2] Cf. Brent Yorgey's explanation of `IO` for the CIS 194 class at UPenn.

Remember: what we have here is a description of how we can get the current time when we need it. We do not have the current time itself yet, so it isn't a value that can be shared, and we don't want it to be shared anyway, because we want it to get a new time whenever we run the program.

And the way we run it is by defining `main` in that module for the runtime system to find and execute. Everything inside `main` is within `IO`, so that everything is nested and sequenced and happens in the order you're expecting.

Another example

Let's look at another example of `IO` turning off sharing. Do you remember the `whnf` and `nf` functions from `criterion` that we used in the last chapter? You may recall that we want to turn off sharing for those, so that they get evaluated over and over again; if the result were shared, our benchmarking would only tell us how long it takes to evaluate it once instead of giving us an average of evaluating it many times. The way we disable sharing for those functions is by applying them to arguments.

But the `IO` variants of those functions do not require this function application in order to disable sharing, because the `IO` parameter itself disables sharing. Contrast the following types:

```
whnf :: (a -> b) -> a -> Benchmarkable
nf :: NFData b
  => (a -> b) -> a -> Benchmarkable

whnfIO :: IO a -> Benchmarkable
nfIO :: NFData a => IO a -> Benchmarkable
```

The `IO` variants don't need a function argument to apply, because sharing is already prevented for IO actions—they can be executed over and over without us having to add an argument.

As we said earlier, `IO` doesn't turn off all sharing everywhere; it couldn't, or else sharing would be meaningless, because `main` is always in `IO`. But it's important to understand when sharing will be disabled and why, because if you've got this notion of sharing running around in the back of your head, you'll have the wrong intuitions for how Haskell code works. Which then leads to...

The code! It doesn't work!

We're going to use an example here that takes advantage of the `MVar` type. This is based on a real code event that was how Chris finally learned what `IO` means and the example he first used to explain it to Julie.

The `MVar` type is a means of synchronizing shared data in Haskell. To give a cursory overview, the `MVar` can hold one value at a time. You put a value into it; it holds onto that value until you take it out. Then, and only then, can you put another value in. We cannot hope to best Simon Marlow's work[3] on this front, so if you want more information about it, we strongly recommend you peruse Marlow's book.

OK, so we'll set up some toy code here with the idea that we want to put a value into an `MVar` and then take it back out:

```
module WhatHappens where

import Control.Concurrent

myData :: IO (MVar Int)
myData = newEmptyMVar

main :: IO ()
main = do
  mv <- myData
  putMVar mv 0
  mv' <- myData
  zero <- takeMVar mv'
  print zero
```

This will spew an error about being stuck or in a deadlock. The problem here is that the type `IO MVar a` of `newEmptyMVar` is a recipe for producing as many empty `MVars` as you need or want; it is not a reference to a single, shared `MVar`. In other words, the two references to `myData` here are not referring to the same `MVar`.

Taking from an empty `MVar` blocks until something is put into the `MVar`. Consider the following ordering:

[3] See Simon Marlow, *Parallel and Concurrent Programming in Haskell*.

```
take
put
take
put
```

That will terminate successfully. An attempt to `take` a value from the `MVar` blocks, then a value is `put` into it, then another blocked `take` occurs, and finally there is another `put` to satisfy the second take. This is fine.

The following is an example of something that will deadlock:

```
put
take
take
```

Whatever part of your program performs the second `take` will now be blocked until a second `put` occurs. If your program is designed such that *no* `put` ever occurs again, it's deadlocked. A deadlock error looks like the following:

```
Prelude> main
*** Exception:
  thread blocked indefinitely
  in an MVar operation
```

When you see a type like:

```
IO String
```

You don't have a `String`; you have a means of (possibly) obtaining a `String`, with some effects *possibly* performed along the way. Similarly, what happened earlier is that we had *two* `MVar`s with two different lifetimes and that looked something like this:

```
mv    mv'
put   take (the final one)
```

The point here is that this type:

```
IO (MVar a)
```

Tells you that you have a recipe for producing as many empty `MVars` as you want, not a reference to a single, shared `MVar`.

You can share the `MVar`, but it has to be done explicitly rather than implicitly. Failing to explicitly share the `MVar` reference after binding it once will simply spew out new, empty `MVars`. Again, we recommend Simon Marlow's book when you're ready to explore `MVars` in more detail.

29.5 IO doesn't disable sharing for everything

As we mentioned earlier, `IO` doesn't disable sharing for everything, and it wouldn't make sense if it did. It only disables sharing for the terminal value it reduces to. Values that are not dependent on `IO` for their evaluation can still be shared, even within a larger IO action such as `main`.

In the following example, we'll use `Debug.Trace` again to show us when things are being shared. For `blah`, the `trace` is outside the IO action, so we'll use `outer trace`:

```
import Debug.Trace

blah :: IO String
blah = return "blah"

blah' = trace "outer trace" blah
```

And for `woot`, we'll use `inner trace` inside the IO action:

```
woot :: IO String
woot = return (trace "inner trace" "woot")
```

Then, we throw both of them into a larger IO action, `main`:

```
main :: IO ()
main = do

  b <- blah'
  putStrLn b
  putStrLn b
```

```
  w <- woot
  putStrLn w
  putStrLn w

Prelude> main
outer trace
blah
blah
inner trace
woot
woot
```

We only see inner and outer emitted once, because `IO` is not intended to disable sharing for values not in `IO` that happen to be used in the course of running of an IO action.

29.6 Purity is losing meaning

It's common at this time to use the words "purely functional" or to talk about *purity* when one means *without effects*. This is inaccurate and not very useful as a definition, but we're going to provide some context here and an alternative understanding.

Semantically, pedantically accurate

Purity and "pure functional" have undergone a few changes in connotation and denotation since the 1950s. What was originally meant when describing a pure functional programming language is that the semantics of the language would only be lambda calculus. For quite a long time, impure functional languages were more typical. They admitted the augmentation of lambda calculus, usually so that the means to describe imperative, effectful programs was embedded within the semantics. The strength of Haskell is that by sticking to lambda calculus, we not only have a much simpler core language, but we retain referential transparency in the language, as well. We use nested lambdas (hidden behind a `Monad` abstraction) to order and encapsulate effects while maintaining referential transparency.

Referential transparency

Referential transparency is something you are probably familiar with, even if you've never called it that before. Put casually, it means that any function, when given the same inputs, returns the same result. More precisely, an expression is referentially transparent when it can be replaced with its value without changing the behavior of a program.

One source of the confusion between *purity* as referential transparency and *purity* as pure lambda calculus could be that in a pure lambda calculus, referential transparency is assured. Thus, a pure lambda calculus is necessarily pure in the other sense, too.

The mistake people make with `IO` is that they conflate the effects with the semantics of the program. A function that returns `IO a` is still referentially transparent, because given the same arguments, it'll generate the same IO action every time! To make this point:

```
module IORefTrans where

import Control.Monad (replicateM)
import System.Random (randomRIO)

gimmeShelter :: Bool -> IO [Int]
gimmeShelter True =
  replicateM 10 (randomRIO (0, 10))
gimmeShelter False = pure [0]
```

The trick here is to realize that while *executing* `IO [Int]` can and does produce different literal values when the argument is `True`, it's still producing the same *result* (i.e., a list of random numbers) for the same input. Referential transparency is preserved, because we're still returning the same *IO action*, or "recipe," for the same argument, the same *means* of obtaining a list of `Int`. Every `True` input to this function will return a list of random `Int`s:

```
Prelude> gimmeShelter True
[1,8,7,9,10,4,2,9,3,6]
Prelude> gimmeShelter True
[10,0,7,1,10,2,4,0,9,3]
Prelude> gimmeShelter False
[0]
```

The sense we're trying to convey here is that as far as Haskell is concerned, it's a language for evaluating expressions and constructing IO actions that get executed by `main` at some point later.

29.7 IO's Functor, Applicative, and Monad

Another mistake people make is in implying that `IO` is a `Monad`, rather than accounting for the fact that, like all `Monads`, `IO` is a datatype that has a `Monad` instance—as well as `Functor` and `Applicative` instances:

`fmap` constructs an action that performs the same effects but transforms an `a` into a `b`:

```
fmap :: (a -> b) -> IO a -> IO b
```

The `<*>` operator constructs an action that performs the effects of both the function and value arguments, applying the function to the value:

```
(<*>) :: IO (a -> b) -> IO a -> IO b
```

`join` merges the effects of a nested IO action:

```
join :: IO (IO a) -> IO a
```

The IO Functor

What does `fmap` mean with respect to `IO`? As always, we want an example:

```
fmap (+1) (randomIO :: IO Int)
```

If we're going to get that `Int` value, we will have to perform some effects. What `fmap` does here is lift our incrementing function over the effects that we might perform to obtain the `Int` value. It doesn't affect the effects, because the effects are part of that `IO` structure. Using `fmap` here returns a recipe for obtaining an `Int` that also increments the result of the original action that is lifted over.

The key is that we don't perform any effects. We produce a new IO action in terms of the old one by transforming the final result of the old one.

Applicative and IO

`IO` also has an `Applicative` instance, as we mentioned in the `Applicative` chapter. You might remember an example like this:

```
Prelude> (++) <$> getLine <*> getLine
hello
julie
"hellojulie"
```

There, we `fmap`'ped the concatenation operator over two (potential) `IO Strings` to produce the final result. Let's look at another, more interesting example:

```
   (+)
<$> (randomIO :: IO Int)
<*> (randomIO :: IO Int)
```

After the initial `fmap`, we have a means of obtaining a function that is monoidally lifted over a means of obtaining an `Int`. In consequence, you'll get a single new means of obtaining the result of having applied the function, which performs the effects of both.

Monad and IO

For `IO`, `pure` or `return` can be read as an effect-free embedding of a value in a recipe-creating environment. Let's consider the following examples.

First, GHCi does basically two things: it can print values not in `IO`, such as these:

```
Prelude> "I'll pile on the candy"
"I'll pile on the candy"
Prelude> 1
1
```

It can also run IO actions and print their results, if any. When you have values of type `IO (IO a)`, what you have is a recipe for making a recipe that produces an `a`. Consider why the following example using `print` does not actually print anything:

```
Prelude> :{
*Main| let embedInIO =
*Main|       return :: a -> IO a
*Main| :}
Prelude> embedInIO 1
1
Prelude> :{
*Main| let s =
*Main|       "I'll put in some ingredients"
*Main| :}
Prelude> embedInIO (print s)
```

In order to merge those effects and get a single `IO a` that will print a result in GHCi, we need `join`:

```
Prelude> s = "It's a piece of cake"
Prelude> join $ embedInIO (print s)
"It's a piece of cake"
Prelude> embedInIO (embedInIO 1)
Prelude> join $ embedInIO (embedInIO 1)
1
```

What sets the `IO Monad` apart from the `Applicative` is that the effects performed by the outer IO action can influence *what* recipe you get in the inner part. The nesting also lets us express order dependence, a useful trick for lambda calculi noted by Peter J. Landin.[4]

An example, for effect:

```
module NestedIO where

import Data.Time.Calendar
import Data.Time.Clock
import System.Random

huehue :: IO (Either (IO Int) (IO ()))
huehue = do
```

[4]See P. J. Landin, *A Correspondence Between ALGOL 60 and Church's Lambda-Notation, Part 1*.

```
  t <- getCurrentTime
  let (_, _, dayOfMonth) =
        toGregorian (utctDay t)

  case even dayOfMonth of
    True ->
      return $ Left randomIO

    False ->
      return $
      Right (putStrLn "no soup for you")
```

The IO action we return here is contingent on having performed effects and observed whether the day of the month was an even number or an odd one. Note that this is inexpressible with `Applicative`. If you'd like a way to run it and see what happens, try the following:

```
Prelude> blah <- huehue
Prelude> either (>>= print) id blah
-7077132465932290066
```

It was the 28th of January when we wrote this. Your mileage may vary.

Monadic associativity

Haskellers will often get confused when they are told that `Monad`'s bind is associative, because they'll think of `IO` as a counterexample. The error being made here is mistaking the construction of IO actions for the *execution* of IO actions. As far as Haskell is concerned, we only construct IO actions to be executed when we call `main`. Semantically, IO actions aren't something we *do*, but something we talk about. Binding over an IO action doesn't execute it, it produces a new IO action in terms of the old one.

You can reconcile yourself with this framing by remembering how IO actions are like recipes, an analogy devised by Brent Yorgey that we're fond of.

29.8 Well, then, how do we MVar?

Earlier in the chapter, we showed you an example of when `IO` prevents sharing, using the `MVar` type. Our previous code would block because the following:

```
myData :: IO (MVar Int)
myData = newEmptyMVar
```

Is an IO action that produces an empty `MVar`; it isn't a stable reference to a single given `MVar`. We have a couple ways of fixing this. One is by passing the single stable reference as an argument. The following will terminate successfully:

```
module WhatHappens where

import Control.Concurrent

main :: IO ()
main = do
  mv <- newEmptyMVar
  putMVar mv (0 :: Int)

  zero <- takeMVar mv
  print zero
```

There is a somewhat more evil and unnecessary way of doing it. We'll use this opportunity to examine an *unsafe* means of enabling sharing for an IO action: `unsafePerformIO`! Consider that the following will also terminate:

```
module WhatHappens where

import Control.Concurrent
import System.IO.Unsafe

myData :: MVar Int
myData = unsafePerformIO newEmptyMVar
```

```
main :: IO ()
main = do
  putMVar myData 0
  zero <- takeMVar myData
  print zero
```

The type of `unsafePerformIO` is `IO a -> a`, which is seemingly impossible and not a good idea in general. In real code, you should pass a reference to an `MVar` as an explicit argument or via `ReaderT`, but the combination of `MVar` and `unsafePerformIO` gives us an opportunity to see in very stark terms what it means to use `unsafePerformIO` in our code. The new empty `MVar` can now be shared implicitly, as often as you want, instead of creating a new one each time.

Do *not* use `unsafePerformIO` when unnecessary, or where it could break referential transparency, in your code! If you aren't sure—don't use it! There are other unsafe `IO` functions, too, but there is rarely a need for any of them, and in general, you should prefer being explicit rather than implicit.

29.9 Chapter exercises

File I/O with Vigenère

Reusing the Vigenère cipher you wrote back in Chapter 11, on algebraic datatypes, and wrote tests for in Chapter 14, on testing, make an executable that takes a key and a mode argument. If the mode is `-d`, the executable decrypts the input from standard in and writes the decrypted text to standard out. If the mode is `-e`, the executable blocks on input from standard input (`stdin`) and writes the encrypted output to standard output (`stdout`).

Consider this an opportunity to learn more about how file handles and the following members of the `base` library work:

```
System.Environment.getArgs :: IO [String]
System.IO.hPutStr
  :: Handle -> String -> IO ()
System.IO.hGetChar :: Handle -> IO Char
System.IO.stdout :: Handle
System.IO.stdin :: Handle
```

Whatever OS you're on, you'll need to learn how to feed files as input to your utility and how to redirect standard output to a file. Part of the exercise is figuring this out for yourself. You'll want to use `hGetChar` more than once to accept an encrypted or decrypted string.

Add timeouts to your utility

Use `hWaitForInput` to make your utility timeout if no input is provided within a span of time of your choosing. You can make it an optional command-line argument. Exit with a nonzero error code and an error message printed to standard error (`stderr`) instead of `stdout`.

```
System.IO.hWaitForInput
  :: Handle -> Int -> IO Bool
System.IO.stderr :: Handle
```

Config directories

Reusing the INI parser that you wrote as an exercise in Chapter 24, on parser combinators, parse a directory of INI config files into a `Map`, the key of which is the filename and the value of which is the result of parsing the INI file. *Only* parse files in the directory that have the file extension `.ini`.

29.10 Follow-up resources

1. Haskell Wiki. *Referential transparency*.
   `https://wiki.haskell.org/Referential_transparency`
2. Haskell Wiki. *IO Inside*.
   `https://wiki.haskell.org/IO_inside`
3. Edward Z. Yang. *Unraveling the mystery of the IO Monad*.
4. Michael Snoyman. *Primitive Haskell*.
5. Haskell Wiki. *Evaluation order and state tokens*.
6. Takenobu Tani. *GHC (STG, Cmm, asm) illustrated for hardware persons*.

7. Simon Peyton Jones. *Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell*.

8. GHC compiler notes. *Note [IO hack in the demand analyser]*.

9. Andrew D. Gordon and Kevin Hammond. *Monadic I/O in Haskell 1.3*.
`https://pdfs.semanticscholar.org/aee2/f74507df6c623cc6863a1db21dca4eefb15f.pdf`

10. Eugenio Moggi. *Notions of computation and monads*.
`http://www.disi.unige.it/person/MoggiE/ftp/ic91.pdf`

11. P. J. Landin. *The Next 700 Programming Languages*.
`https://www.cs.cmu.edu/~crary/819-f09/Landin66.pdf`

12. Haskell Report 1.2
`http://haskell.org/definition/haskell-report-1.2.ps.gz`

Chapter 30

When Things Go Wrong

> It is easier to write an incorrect program than understand a correct one.
>
> Alan Perlis

30.1 Exceptions

Let's face it: in the execution of a program, a lot of things can happen, not all of them expected or desired. In those unhappy times, when things have not gone as we wanted them to, we will throw or raise an exception. The term *exception* refers to a condition that has interrupted the expected execution of a program. Encountering an exception causes an error, or exception, message to appear, informing you that due to some condition you weren't prepared for, the execution of the program has halted in an unfortunate way.

In previous chapters, we covered ways of using `Maybe`, `Either`, and `Validation` types to handle certain error conditions explicitly. Raising exceptional conditions via such datatypes isn't always ideal, however. In some cases, exceptions can be faster, by eliding repeated checks for an adverse condition. Exceptions are not explicitly part of the interfaces you're using, and that has immediate consequences when trying to reason about the ways in which your program could fail.

Letting exceptions arise as they will—and the program halt willy-nilly—is suboptimal. Exception handling is a way of dealing with errors and giving the program some alternate means of execution or termination should one arise. This chapter is going to cover both exceptions and what they look like as well as various means of handling them.

In this chapter, we will:

- Examine the `Exception` type class and methods.
- Dip our toes into existential quantification.
- Discuss ways of handling exceptions.

30.2 The Exception class and methods

Exceptions are plain old types and values like you've seen throughout the book. The types that encode exceptions, however, must have an instance of the `Exception` type class. The origins of exceptions as they exist in Haskell today are in Simon Marlow's work on an extensible hierarchy of exceptions that are discriminated at runtime.[1] Using

[1] `http://community.haskell.org/~simonmar/papers/ext-exceptions.pdf`

this extensible hierarchy allows you to both catch exceptions that may have various types and also add new exception types as the need arises.

The `Exception` type class definition looks like this:

```
class (Typeable e, Show e) =>
      Exception e where
  toException :: e -> SomeException
  fromException :: SomeException -> Maybe e
  displayException :: e -> String
  -- Defined in GHC.Exception
```

We'll take a look at these methods in a moment. The `Show` constraint is there so that we can print the exception to the screen in a readable form for whatever type `e` ends up being. `Typeable` is a type class that defines methods for identifying types at runtime. We will talk about this more and explain why these constraints are necessary to our `Exception` class soon.

The list of types that have an `Exception` instance is long:

```
-- some instances elided
instance Exception IOException
instance Exception Deadlock
instance Exception BlockedIndefinitelyOnSTM

instance
  Exception BlockedIndefinitelyOnMVar
instance Exception AsyncException
instance Exception AssertionFailed

instance Exception AllocationLimitExceeded
instance Exception SomeException
instance Exception ErrorCall
instance Exception ArithException
```

We won't talk in detail about each of these, but you may be able to figure out what, for example, `BlockedIndefinitelyOnMVar` is used for. We'll note that it's simply a datatype with one inhabitant:

```
data BlockedIndefinitelyOnMVar =
  BlockedIndefinitelyOnMVar
  -- Defined in GHC.IO.Exception
```

If we look at `ArithException`, we'll find that it's a sum type with several values:

```
data ArithException
  = Overflow
  | Underflow
  | LossOfPrecision

  | DivideByZero
  | Denormal
  | RatioZeroDenominator

instance Exception ArithException
```

If you import the `Control.Exception` module, you can poke at `ArithException`'s data constructors and see that they're plain old values, nothing unusual at all.

But there is something different going on here

We're going to start unpacking of all this to see how the parts work together. First, let's take a look at the methods of the `Exception` type class:

```
toException   :: e -> SomeException
fromException :: SomeException -> Maybe e
```

We don't have much occasion to use the `toException` and `fromException` functions themselves. Instead, we use other functions that call them for us. As it turns out, the `toException` method is quite similar to the data constructor for `SomeException`. You may have noticed that `SomeException` is also a type that is listed as having an instance of the `Exception` type class, and now here it is in the `Exception` methods. It seems a bit circular, but it turns out that `SomeException` is ultimately the key to the way we handle exceptions.

A brief introduction to existential quantification

`SomeException` acts as a sort of parent type for all the other exception types, so that we can handle many exception types at once, without having to match all of them. Let's examine how:

```
data SomeException where
  SomeException
    :: Exception e => e -> SomeException
```

This may not seem odd at first glance. That is due, in part, to the fact that the weirdness is hiding in a construction called a GADT, short for generalized algebraic datatype. For the most part, GADTs are out of the scope of this book, being well into intermediate Haskell territory that is fun to explore but not strictly necessary for programming in Haskell. What the GADT syntax is hiding there is something called *existential quantification*.

We could rewrite the `SomeException` type like this, without a change in meaning:

```
data SomeException =
  forall e . Exception e => SomeException e
```

Ordinarily, the `forall` quantifies variables universally, as you might guess from the word *all*. However, the `SomeException` type constructor doesn't take an argument; the type variable `e` is a parameter of the *data constructor*. It takes an `e` and results in a `SomeException`. Moving the quantifier to the data constructor limits the scope of its application and changes the meaning from *for all e* to *there exists some e*. That is existential quantification. It means that any type that implements the `Exception` class can be that `e` and be subsumed under the `SomeException` type.

We aren't going to examine existential quantification deeply here; this is a mere taste. Usually, when type constructors are parameterized, they are universally quantified. Arguments have to be supplied to satisfy them. Your `Maybe a` type is, as we've noted before, a sort of function waiting for an argument to be supplied to be a fully realized type.

But when we existentially quantify a type, as with `SomeException`, we can't do much with that polymorphic type variable in its data con-

structor. That is, we can't concretize it. Other than adding constraints, we can't know anything about it. It *must* remain polymorphic, and we can cram any value of any type that implements its constraint into that role. It's like a polymorphic parasite just hanging out on your type.

So, any exception type—any type with an instance of the `Exception` type class—can be that `e` and be handled as a `SomeException`. We need `Typeable` and `Show` in order to determine what type of exception we're dealing with, as we will soon see.

So, wait, what?

For an example of what existential quantification lets us do, we're going to show you an example that doesn't rely on the magic of the runtime exception machinery. Here, we'll be returning errors in `Either` of totally different types without having to unify them under a single sum type:

```
{-# LANGUAGE ExistentialQuantification #-}
{-# LANGUAGE GADTs                     #-}

module WhySomeException where

import Control.Exception
  ( ArithException(..)
  , AsyncException(..))
import Data.Typeable

data MyException =
  forall e .
  (Show e, Typeable e) => MyException e

instance Show MyException where
    showsPrec p (MyException e) =
      showsPrec p e

multiError :: Int
           -> Either MyException Int
multiError n =
```

```
  case n of
    0 ->
      Left (MyException DivideByZero)

    1 ->
      Left (MyException StackOverflow)
    _ -> Right n
```

What's special about the code above is that we have a `Left` case in our `Either` that includes error values of two totally different types without enumerating them in a sum type. `MyException` doesn't appear to have a polymorphic argument in the type constructor, but it does in the data constructor. We are able to apply the `MyException` data constructor to values of different types because of the existentially quantified type for `e`:

```
data SomeError =
    Arith ArithException
  | Async AsyncException
  | SomethingElse
  deriving (Show)

discriminateError :: MyException
                  -> SomeError
discriminateError (MyException e) =

  case cast e of
    (Just arith) -> Arith arith

    Nothing ->
      case cast e of
        (Just async) -> Async async
        Nothing -> SomethingElse

runDisc n =
  either discriminateError
  (const SomethingElse) (multiError n)
```

Then, we try this out:

```
Prelude> runDisc 0
Arith divide by zero
Prelude> runDisc 1
Async stack overflow
Prelude> runDisc 2
SomethingElse
```

This is the essence of why we need existential quantification for exceptions—so that we can throw various exception types without being forced to centralize and unify them under a sum type. Don't abuse this facility.

Prior to this design, there were a few different ways you could do exception handling in Haskell. Two of the more apparent methods would have been one big sum type or just using strings. The problem is that neither of them are meaningfully extensible to structured, proper datatypes. We want, in a sense, a hierarchy of values where catching a "parent" means catching any of its possible "children." The combination of `SomeException` and the `Typeable` type class gives you a means of throwing different exceptions of different types and then catching some or all of them in a handler without wrapping them in a sum type.

Typeable

The `Typeable` type class lives in the `Data.Typeable` module. `Typeable` exists to permit types to be known at runtime, allowing for a sort of dynamic type checking. It allows you to learn the type of a value at runtime and also to compare the types of two values and check that they are the same. `Typeable` is particularly useful when you have code that needs to allow various types to be passed to it but needs to enforce or trigger on specific types. Note that since GHC 8.0 you will not need to explicitly derive `Typeable` on your datatypes in order to use the `Data.Typeable` API.

This is ordinarily unwise, but it makes sense when you're talking about exceptions. When we're concerned with exception handling, we want to be able to check whether values of possibly varying types match the `Exception` type we're trying to handle, and we need to do that at runtime, when the exception occurs. Thus, we need this runtime witness to the types of the exceptions.

Let's look at a method called `cast`, simplified from its implementation in `base`:

```
cast :: (Typeable a, Typeable b)
     => a -> Maybe b
```

We don't usually call this function directly, but it gets called for us by the `fromException` function, and `fromException` is called by the `catch` function.

At runtime, when an exception is thrown, it starts rolling back through the call stack, looking for a `catch`. When it finds a `catch`, it checks to see what type of exception this `catch` catches. It calls `fromException` and `cast` to check if the type of the exception that got thrown matches the type of an exception we're handling with the `catch`. A `catch` that handles a `SomeException` will match any type of exception, due to the flexibility of that type.

If they don't match, we get a `Nothing` value; the exception will keep moving through the stack, looking for a `catch` that can handle the exception that was thrown. If it doesn't find one, your program just dies an unseemly death.

If they do match, the `Just a` allows us to catch the exception.

30.3 This machine kills programs

Exceptions can result from pure code:

```
Prelude> 2 `div` 0
*** Exception: divide by zero
```

However, running code is an IO action (and GHCi is implicitly invoking `IO`), so most of the time when you need to worry about exceptions, you'll be in `IO`. Even when they happen in pure code, exceptions may only be caught, or handled, in `IO`.

`IO` contains the implicit contract, "You cannot expect this computation to succeed unconditionally." It turns out the outside world is a harsh mistress—just about any IO action can fail, even `putStrLn`.

First, let's demonstrate that any IO action can fail. We will assume that you do not currently have a file called `aaa` in your working directory. So, when you run this code, it will create the file, write to it, print `"wrote to file"` in your terminal, and terminate successfully:

```
module Main where

main = do
 writeFile "aaa" "hi"
 putStrLn "wrote to file"
```

You can fire up your REPL and load that code, or you can compile the binary like this (this is review, so if you already have all this down, then go ahead and do it):

```
stack ghc -- <filename> -o <output file name>
```

Run it like this:

```
$ ./<output file name>
```

If you call the output file `wp`, for example, your terminal session might look like this:

```
$ stack ghc -- writepls.hs -o wp
[stack compilation messages]
$ ./wp
wrote to file
$ cat aaa
hi
```

Cool, that all worked. It worked, in part, because `writeFile` will go ahead and create a file and give it write permissions if the file you're trying to write to does not exist. But what if you're trying to write to a file that does already exist and does not have write permissions?

Make a read-only file named `zzz` that we can experiment with. To make a file that cannot be written to on Linux or macOS, the following suffices:

```
$ touch zzz
$ chmod 400 zzz
```

Suppose that file inhabits a directory where we're trying to execute this program:

```
module Main where
```

```
main = do
 writeFile "zzz" "hi"
 putStrLn "wrote to file"
```

It's the same program we had for the `aaa` file, just with the file name changed. You can fire up your REPL and load that, or you can compile the binary as we did above.

Then, if you run this program with such a file, you'll get the following result:

```
$ ./wp
wp: zzz: openFile: permission denied (Permission denied)
```

There's a hole in our bucket, dear Liza: an exception.

Catch me if you can

Let's fix that, dear Henry. We'll start with some rudimentary exception handling:

```
module Main where

import Control.Exception
import Data.Typeable

handler :: SomeException -> IO ()
handler (SomeException e) = do
  print (typeOf e)
  putStrLn ("We errored! It was: "
            ++ show e)

main =
  writeFile "zzz" "hi"
    `catch` handler
```

We're still going to terminate without writing to the file, for the same reasons as above. The program will run and terminate successfully, but it'll mention the error and say that it failed with an `IOException`. We'll get a bit more information about why the program failed and be able to log that information with our exception handler if we wish. Sometimes, that's exactly what you want: for your

program to log the exception and then die. Soon, we'll look at some other options for handling exceptions in a way that lets your program proceed with an alternative execution.

For now, let's turn our attention to `catch`:

```
catch :: Exception e
      => IO a
      -> (e -> IO a)
      -> IO a
```

You may recall that we mentioned `catch` earlier, because it calls `fromException` and `cast` for us. It runs only if the exception matching the type you specify gets thrown, and it gives you an opportunity to recover from the error and still satisfy the original type that your IO action purports to be. If no exception gets thrown, then nothing happens with that `e`, and the `IO a` at the front is the same as the `IO a` at the end.

Let's expand our rudimentary error handling in a way that provides the program an alternative execution method instead of just allowing it to die. This time, the `main` action still wants to write to that read-only file, but our handler gives it an alternate file that does not exist to write to (if you do have a file called `bbb` in your present working directory, you can change the name of the `writeFile` argument to some other name, anything, as long as it doesn't exist in your directory yet):

```
module Main where

import Control.Exception
import Data.Typeable

handler :: SomeException -> IO ()
handler (SomeException e) = do
  putStrLn ("Running main caused an error!\
           \ It was: "
            ++ show e)
  writeFile "bbb" "hi"
```

```
main =
  writeFile "zzz" "hi"
    `catch` handler
```

When writing to `zzz` fails, it should print the error message to the terminal. If you check your directory, you should see your alternative file, named in the handler function, and if you look inside it, it should say `"hi"` to you.

Let's look at another, slightly more complex, use of `catch`. This is taken from a program that deletes things from a Twitter account and relies on the library `twitter-conduit`.[2] This portion of the program can fail when it doesn't have access to the appropriate credentials for talking to a Twitter account. So, we build an exception handler that tells it what to do when that exception arises:

```
withCredentials action = do
  twinfo <-
    loadCredentials `catch` handleMissing

  case twinfo of
    Nothing     ->
      getTWInfo >>= saveCredentials
    Just twinfo -> action twinfo

  where handleMissing :: IOException
                      -> IO (Maybe TWInfo)
        handleMissing _ = return Nothing
```

We turn an `IOException` into an `IO (Maybe a)`, so we can case on the `Maybe` to tell it what to do in the `Nothing` case. Now, if we throw an `IOException` and return a `Nothing` value, our program will execute this:

```
getTWInfo >>= saveCredentials
```

By saving the credentials (the code that does the saving is not shown here), we should not encounter this exception the next time we try to run the program. And so, we perform the `action` that is named in the `Just twinfo` line (said action is also not shown here—sorry).

[2] `https://www.stackage.org/package/twitter-conduit`

30.4 Want either? Try!

Sometimes, we'd like to lift exceptions out into explicit `Either` values. This is quite doable, but you can't erase the fact that you performed I/O in the process. It's also no guarantee you'll catch every exception. Here's the function we need to turn an implicit exception into an explicit `Either`:

```
-- Control.Exception
try :: Exception e
    => IO a
    -> IO (Either e a)
```

To use it, we could write something like the following code (please note, this will not compile to a binary the way earlier examples do, because it is not a `Main` executable; therefore, use GHCi):

```
module TryExcept where

import Control.Exception

willIFail :: Integer
          -> IO (Either ArithException ())
willIFail denom =
  try $ print $ div 5 denom
```

Here, we print the result, because you can only handle exceptions in `IO`, evidenced by the types of `try` and `catch`. If you feed this some inputs, you'll see something like the following:

```
Prelude> willIFail 1
5
Right ()
Prelude> willIFail 0
Left divide by zero
```

One thing to keep in mind is that exceptions in Haskell are like exceptions in most other programming languages—they are *imprecise*. An exception unwinds the stack until a piece of code catches the exception or kills your whole program.

If you want to get rid of the `Right ()` that it's printing in the successful cases, here's one way to do so:

```
onlyReportError :: Show e
                => IO (Either e a)
                -> IO ()
onlyReportError action = do

  result <- action
  case result of
    Left e -> print e
    Right _ -> return ()

willFail :: Integer -> IO ()
willFail denom =
  onlyReportError $ willIFail denom
```

Or you could use `catch`:

```
willIFail' :: Integer -> IO ()
willIFail' denom =
  print (div 5 denom) `catch` handler
    where handler :: ArithException
                  -> IO ()
          handler e = print e
```

Let's expand on this. We want to take the above examples and turn them into an executable binary, which is a problem, because in an executable, `main` can't take arguments. So, we'll have to do some serious modification in order to be able to pass arguments to `main` when we call it. We're going to import `System.Environment` so that we can make use of a function called `getArgs` that allows us to pass arguments in at the point where we call `main`:

```
module Main where

import Control.Exception
import System.Environment (getArgs)

willIFail :: Integer
          -> IO (Either ArithException ())
willIFail denom =
  try $ print $ div 5 denom
```

```
onlyReportError :: Show e
                => IO (Either e a)
                -> IO ()
onlyReportError action = do
  result <- action
  case result of
    Left e -> print e
    Right _ -> return ()

testDiv :: String -> IO ()
testDiv d =
  onlyReportError $ willIFail (read d)

main :: IO ()
main = do
  args <- getArgs
  mapM_ testDiv args
```

The usage of `mapM_` here might not be obvious, so let's unpack that a bit. It is essentially a less general `traverse` function that throws away its end result and only produces the effects. In this case, those effects are going to be the results of mapping our `testDiv` function over a list of arguments, returning either the result of a successful division or the type of an exception.

We'll compile this one to an executable binary again, as we did earlier in the chapter. To pass in the arguments, it will look like this:

```
$ stack ghc -- writepls.hs -o wp
...stack noise...
$ ./wp 4 5 0 9
1
1
divide by zero
0
```

In case you want to try this in the REPL, reproducing what you did above, use the GHCi `:main` command and pass the same arguments:

```
Prelude> :main 4 5 0 9
```

```
1
1
divide by zero
0
```

Notice that, now that the exception is handled, we can still get that last result—we have survived an `ArithException`!

30.5 The unbearable imprecision of trying

Let's do another little experiment:

```
import Control.Exception

canICatch :: Exception e
          => e
          -> IO (Either ArithException ())
canICatch e =
  try $ throwIO e
```

The new thing here is `throwIO`, a function that allows you to throw an exception. Right now, we want to demonstrate that this handler doesn't catch all types of exceptions, so we're using `throwIO` to cause exceptions of various types to be thrown.

The `Left` here can only handle or catch an `ArithException`, not any other kind. So when we throw a different type of exception, we get the following result:

```
Prelude> canICatch DivideByZero
Left divide by zero
Prelude> canICatch StackOverflow
*** Exception: stack overflow
Prelude> :t DivideByZero
DivideByZero :: ArithException
Prelude> :t StackOverflow
StackOverflow :: AsyncException
```

The latter case blows past our `try`, because we are trying to catch an `ArithException`, not an `AsyncException`.

We've mentioned several times that `SomeException` will match on all types that implement the `Exception` type class, so try rewriting the above so that the `StackOverflow` or any other exception can also be caught.

We'll continue the experiment by making a program that runs until an unhandled exception stops the party:

```
module StoppingTheParty where

import Control.Concurrent (threadDelay)
import Control.Exception
import Control.Monad (forever)
import System.Random (randomRIO)

randomException :: IO ()
randomException = do
  i <- randomRIO (1, 10 :: Int)
  if i `elem` [1..9]
    then throwIO DivideByZero
    else throwIO StackOverflow

main :: IO ()
main = forever $ do
  let tryS :: IO ()
           -> IO (Either ArithException ())
      tryS = try

  _ <- tryS randomException
  putStrLn "Live to loop another day!"
  -- microseconds
  threadDelay (1 * 1000000)
```

We've talked about `forever` before; it causes the program execution to loop indefinitely. We have added the `threadDelay` to slow the looping down, so that what's happening is more noticeable. Note that the thread is delayed by a number of microseconds.

The `tryS` allows it to survive the `ArithExceptions`. We throw away those exceptions and keep looping, but we can only throw away the exception that we match on (`ArithException`). At some point, when

our random number is 10, we will throw an `AsyncException` instead of an `ArithException`, and our program will die a rapid death.

Try modifying this one, so that both exceptions are handled and the loop never terminates.

30.6 Why throwIO?

It may have seemed odd to you (or not!) to encounter `throwIO` above. Why do we want to stop a program by purposely throwing an exception? In the real world, we often do want to do that—to stop the program when some condition occurs, but it may be difficult to see that from what we've shown you so far.

There's a function called `throw` that throws exceptions without `IO`. The arithmetic exceptions are thrown by this function. It's what the `div` function uses to throw a `DivideByZero` exception. You don't need `throw` and should never use it. Use `throwIO` and similar, instead.

The difference between `throw` and `throwIO` can be seen in the types:

```
throw :: Exception e => e -> a
throwIO :: Exception e => e -> IO a
```

Partiality in the form of throwing an exception can be thought of as an effect. The conventional way to throw an exception is to use `throwIO`, which has `IO` in its result. This is the same thing as `throw`, but `throwIO` embeds the exception in `IO`. You always handle exceptions in `IO`.[3] Handling exceptions must be done in `IO`, even if they were thrown without an `IO` type. You almost never want `throw`, as it throws exceptions without any warning in the type, even `IO`.

We'll look at an example of an unconditionally thrown exception in `IO`, so you can see how it affects the control flow of your program:

```
import Control.Exception

main :: IO ()
main = do
  throwIO DivideByZero
  putStrLn "lol"
```

[3]Why? Because catching and handling exceptions means you could produce different results from the same inputs. That breaks referential transparency.

```
Prelude> main
*** Exception: divide by zero
```

Like `throw`, `throwIO` is often called for us, behind the scenes, by library functions. Often, in interacting with the real world, we need to tell our program that in certain conditions, we want it to stop or to give us an error message and let us know things went wrong. We'll take a look at a couple of examples from real code, in this case from a library called `http-client`[4] by Michael Snoyman, that uses `throwIO` to throw some exceptions when some `http` things haven't gone the way we wanted them to:

```
connectionReadLine :: Connection
                   -> IO ByteString
connectionReadLine conn = do
    bs <- connectionRead conn
    when (S.null bs) $
      throwIO IncompleteHeaders
    connectionReadLineWith conn bs
```

In the above, `throwIO` will throw an `IncompleteHeaders` exception when the `ByteString` header is empty. In the next example, it's used to throw a `ResponseTimeout` exception when, well, the response times out:

```
parseStatusHeaders :: Connection
                   -> Maybe Int
                   -> Maybe (IO ())
                   -> IO StatusHeaders
parseStatusHeaders conn timeout' cont
    | Just k <- cont =
        getStatusExpectContinue k
    | otherwise =
        getStatus

  where
    withTimeout = case timeout' of
        Nothing -> id
        Just  t ->
```

[4] `https://www.stackage.org/package/http-client`

```
        timeout t >=>
        maybe
          (throwIO ResponseTimeout)
          return
  -- ...other code elided...
```

You can use `http-client` without worrying about how it makes the exceptions happen. But let's next take a look at making our own exception types for those times when you do need to worry about it. Keep in mind that since the time of writing, `http-client` has changed how it defines and throws exceptions, but the examples should still be useful.

30.7 Making our own exception types

Often, we'll want our own exception types, like `http-client` has. They enable us to be more precise about what's going on in our program. Let's work through a small example to emit one of a couple different possible errors in an otherwise simple function, to see how we could do this:

```
module OurExceptions where

import Control.Exception

data NotDivThree =
  NotDivThree
  deriving (Eq, Show)

instance Exception NotDivThree

data NotEven =
  NotEven
  deriving (Eq, Show)

-- special syntax for automatically
-- deriving Exception
instance Exception NotEven
```

Note here that `Exception` instances are derivable—you don't need to write them yourself. Continuing on:

```
evenAndThreeDiv :: Int -> IO Int
evenAndThreeDiv i
  | rem i 3 /= 0 = throwIO NotDivThree
  | odd i = throwIO NotEven
  | otherwise = return i
```

Then, we'll see the error and success conditions:

```
*OurExceptions> evenAndThreeDiv 0
0
*OurExceptions> evenAndThreeDiv 1
*** Exception: NotDivThree
*OurExceptions> evenAndThreeDiv 2
*** Exception: NotDivThree
*OurExceptions> evenAndThreeDiv 3
*** Exception: NotEven
*OurExceptions> evenAndThreeDiv 6
6
*OurExceptions> evenAndThreeDiv 9
*** Exception: NotEven
*OurExceptions> evenAndThreeDiv 12
12
```

There is an issue with this setup, although it's common. What if we want to know what input or inputs cause the error? We need to add context!

Adding context

Convenient subsection titling! Anyhow, let's modify that last program:

```
module OurExceptions where

import Control.Exception

data NotDivThree =
  NotDivThree Int
  deriving (Eq, Show)

instance Exception NotDivThree
```

```
data NotEven =
  NotEven Int
  deriving (Eq, Show)

instance Exception NotEven

evenAndThreeDiv :: Int -> IO Int
evenAndThreeDiv i
  | rem i 3 /= 0 = throwIO (NotDivThree i)
  | odd i = throwIO (NotEven i)
  | otherwise = return i
```

Now, when we get errors, we will know what input causes them:

```
*OurExceptions> evenAndThreeDiv 12
12
*OurExceptions> evenAndThreeDiv 9
*** Exception: NotEven 9
*OurExceptions> evenAndThreeDiv 8
*** Exception: NotDivThree 8
*OurExceptions> evenAndThreeDiv 3
*** Exception: NotEven 3
*OurExceptions> evenAndThreeDiv 2
*** Exception: NotDivThree 2
```

Catch one, catch all

You can probably figure out how to catch these two, different errors:

```
catchNotDivThree :: IO Int
                 -> (NotDivThree -> IO Int)
                 -> IO Int
catchNotDivThree = catch

catchNotEven :: IO Int
             -> (NotEven -> IO Int)
             -> IO Int
catchNotEven = catch
```

Or perhaps with `try`:

```
Prelude> type EA e = IO (Either e Int)
Prelude> try (evenAndThreeDiv 2) :: EA NotEven
*** Exception: NotDivThree 2
Prelude> try (evenAndThreeDiv 2) :: EA NotDivThree
Left (NotDivThree 2)
```

The type synonym isn't semantically important, but it shrinks the noise a bit. Now, you *could* handle both errors with the `catches` function:

```
catches :: IO a -> [Handler a] -> IO a

catchBoth :: IO Int
             -> IO Int
catchBoth ioInt =

  catches ioInt
  [ Handler
      (\(NotEven _) -> return maxBound)
  , Handler
      (\(NotDivThree _) -> return minBound)
  ]
```

The `maxBound`/`minBound` thing is not good code for real use, just a convenience. Incidentally, the same trick the `SomeException` type uses to hide type arguments is used by the `Handler` type to wrap the values in the list of exception handlers—existential quantification:

```
data Handler a where
  Handler :: Exception e
          => (e -> IO a) -> Handler a
          -- Defined in Control.Exception
```

We can make a list of handlers that handle exceptions of *varying types*, because the exception types are existentially quantified under the datatype of `Handler`.

But what if this isn't convenient enough? What if we have a family of semantically related or otherwise similar exceptions we want to catch as a group? For this, we revive our old friend, the sum type!

```
module OurExceptions where

import Control.Exception

data EATD =
    NotEven Int
  | NotDivThree Int
  deriving (Eq, Show)

instance Exception EATD

evenAndThreeDiv :: Int -> IO Int
evenAndThreeDiv i
  | rem i 3 /= 0 = throwIO (NotDivThree i)
  | even i = throwIO (NotEven i)
  | otherwise = return i
```

Now, when we want to catch either error, we only need one handler, and then we can pattern match on the exception type just like good, old-fashioned datatypes:

```
Prelude> type EA e = IO (Either e Int)
Prelude> try (evenAndThreeDiv 0) :: EA EATD
Left (NotEven 0)
Prelude> try (evenAndThreeDiv 1) :: EA EATD
Left (NotDivThree 1)
```

Nifty, eh? The notion here is to exercise the same taste and judgment in designing your error types as you would in your happy-path types. Preserve context, and try to make it so somebody could understand the problem you're solving from the types. If necessary. On a desert island. With a lot of rum.

And sea turtles.

30.8 Surprising interaction with bottom

One thing to watch out for is a situation in which you catch an exception for a value that might be bottom. Due to non-strictness, the bottom could be forced before or after your exception handler, so you might be surprised if you expect either:

- That your exception handler is meant to catch the bottom.
- That no bottoms would cause your program to fail after it catches, say, a `SomeException`.

The proper coping mechanism for this is a glass of scotch and to realize the following things:

- The exception handling mechanism is not for, nor should be used for, catching bottoms.
- Having caught an exception, even `SomeException`, without rethrowing an exception, doesn't mean your program won't fail.

To demonstrate these points, we'll show you a case in which we catch an exception from a bottom and a case where a bottom leapfrogs our handler:

```
import Control.Exception

noWhammies :: IO (Either SomeException ())
noWhammies =
  try undefined

megaButtums :: IO (Either SomeException ())
megaButtums =
  try $ return undefined
```

Do you think these should have the same result? We've got bad news:

```
Prelude> noWhammies
Left Prelude.undefined
Prelude> megaButtums
Right *** Exception: Prelude.undefined
```

The issue here is that non-strictness means that burying the bottom in a `return` prevents it from being forced until you're already *past* the `try`, resulting in an uncaught error inside the `Right` constructor. The takeaway here shouldn't be "laziness is terrifying" but rather "write total programs that don't use bottom." It's not only unforced bottoms that can cause programs that shouldn't have any uncaught exceptions to fail, either, there are also...

30.9 Asynchronous exceptions

Asynchronous exceptions are the predators hunting your happy little programs. You probably don't have much experience with anything like this, unless you've written Erlang before. Even then, Erlang's asynchronous exceptions are handled by a separate process. Most languages don't have anything like this, if only because they don't have any hope of making it safe within their implementation runtimes:

```
module Main where

import Control.Concurrent
  (forkIO, threadDelay)

import Control.Exception
import System.IO

openAndWrite :: IO ()
openAndWrite = do

  h <- openFile "test.dat" WriteMode
  -- You may need to jiggle this
  threadDelay 1500

  hPutStr h
    (replicate 100000000 '0' ++ "abc")
  hClose h

data PleaseDie =
  PleaseDie
  deriving Show

instance Exception PleaseDie

main :: IO ()
main = do
  threadId <- forkIO openAndWrite
  threadDelay 1000
  throwTo threadId PleaseDie
```

If you run this program, the intended result is that you'll have a file named `test.dat` with only zeroes that doesn't reach the `"abc"` at the end. Since we can't predict the future, if you have a disk with preternaturally fast I/O, increase the arguments to `replicate` to reproduce the intended issue. If it ain't broken, break it.

What happens is that we throw an asynchronous exception from the main thread to our child thread, short-circuiting what we are doing in the middle of doing it. If you do this in a loop, you'll leak file handles, too. Done continually over a period of time, leaking file handles can cause your process to get killed or your computer to become unstable.

We can think of asynchronous exceptions as exceptions raised from a different thread than the one that receives the error. They're immensely useful and give us a means of talking about error conditions that are quite real and possible in languages that don't have formal asynchronous exceptions. Your process can get axe-murdered by the operating system out of nowhere in any language. We just happen to have the ability to do the same within the programming language at the thread level, as well. The issue is that we want to temporarily ignore exceptions until we've finished what we're doing. This is so the state of the file is correct but also so that we don't leak resources like file handles or perhaps database connections or something similar.[5] Never fear, we can fix this!

```
module Main where

-- We haven't explained this.
-- Tough cookies.
import Control.Concurrent
  (forkIO, threadDelay)
import Control.Exception
import System.IO

openAndWrite :: IO ()
openAndWrite = do
  h <- openFile "test.dat" AppendMode
  threadDelay 1500
```

[5] In this case, *leaking* means having too many files, database connections, etc. open at one time, thus consuming all the resources your OS can allocate, the way trying to hold too much in memory for too long causes *memory leaks*.

```
  hPutStr h
    (replicate 10000000 '0' ++ "abc")
  hClose h

data PleaseDie =
  PleaseDie
  deriving Show

instance Exception PleaseDie

main :: IO ()
main = do
  threadId <- forkIO (mask_ openAndWrite)
  threadDelay 1000
  throwTo threadId PleaseDie
```

Here, we use `mask_` from `Control.Exception` in order to mask or delay exceptions thrown to our child thread until the IO action `openAndWrite` is complete. Incidentally, since the end of the mask is the last thing our child thread does, the exception our main thread tries to throw to the child blows up in its face, Wile E. Coyote style, and is now thrown within the main thread.

Don't panic!

Async exceptions are helpful and manifest in less obvious ways in other language runtimes and ecosystems. Don't try to catch everything; just let it die, and make sure you have a process supervisor and good logs. No execution is better than bad execution.

30.10 Follow-up resources

1. Erin Swenson-Healey. *A Beginner's Guide to Exceptions in Haskell*. `https://www.youtube.com/watch?v=PWS0Whf6-wc`
2. Simon Marlow. *Parallel and Concurrent Programming in Haskell*. See Chapter 8, "Overlapping Input/Output" and Chapter 9, "Cancellation and Timeouts." `https://simonmar.github.io/pages/pcph.html`

3. Simon Marlow. *An Extensible Dynamically-Typed Hierarchy of Exceptions*.
https://simonmar.github.io/bib/papers/ext-exceptions.pdf

Chapter 31

Final Project

31.1 Final project

For our final project, we're doing something a little weird, but small and modernized a bit from the original design. Surely no one who knows us from Twitter or IRC will be surprised that we've chosen something eccentric for this, but we felt it was important to show you an end-to-end project that brings in so much real world, it'll make your head spin.

In this chapter:

- FINGER DAEMONS.

31.2 fingerd

Dating back to 1971, finger[1] was a means of figuring out how to contact colleagues or other people on the same computer network and whether they were on the network at a given time, often on the same mainframe, at a time when computing was usually time-shared on the same physical machine. The finger service was originally intended to be used to share an office number, email address—basic contact details like that. By the 1990s, when public internet access became widely available, finger was also used to deliver `.plan` or `.project` files as sort of pre-Twitter/Tumblr microblog.

We're going to be writing a *finger daemon* in this chapter. Finger daemon programs are often called `fingerd`. A *daemon* is a process that runs in the background without direct user interaction; in the case of finger, the daemon acts as the server side of the protocol, while the `finger` program itself is on the client side. When you use `finger` from your command line, it sends a request to the finger daemon, and the daemon responds with the requested information, if it can.

We use this as an example, in part, because it's not a typical web app, only requires working with text, and because the text-based protocol is spare and easy to debug once you know how.

Caveat for Windows users

This chapter is going to be somewhat more Unix/Linux-oriented than previous ones, for a few reasons. Windows users will find that

[1] `http://www.rajivshah.com/Case_Studies/Finger/Finger.htm`

they cannot follow along with all of the examples literally, but the final version of the finger daemon should work.

Even though you will not be able to follow all of the instructions here verbatim, you can still build and hack on the project. If you aren't willing to install a finger client for testing your finger daemon via Cygwin, however, then you'll need to write your own client.

31.3 Exploring finger

If you have `fingerd` running on your local machine under the username `callen`, the result of executing it might look something like this:

```
$ finger callen@localhost
Login: callen               Name: callen
Directory: /home/callen     Shell: /bin/zsh
```

On macOS, this will work even if you haven't fired up or installed a finger service, without specifying a hostname to query:

```
$ finger callen
Login: callen             Name: Chris Allen
Directory: /Users/callen  Shell: /bin/bash
```

Spooky! Don't ask. The finger protocol operates over Transmission Control Protocol (TCP) sockets, something it has in common with the protocol used by web browsers. However, while they both use TCP, a finger daemon is not a web server. It's something much simpler. Rather than having an entire application protocol layered atop TCP like the web (HTTP) does, it's a single message text protocol. Rather than go into a long explanation of the internet, UDP, and TCP, let's say TCP is a protocol for sending messages back and forth between a client and a server. Those messages can be raw bytes or text. A *socket* is an address where a message can be delivered.[2]

Leaving aside the socket business, the way this should work is roughly like this: the client requests some information, and that request is transmitted to the server with TCP magic. The server (our

[2]If you're new to networking and sockets, this guide by Julia Evans is a great introduction: `https://wizardzines.com/zines/networking/`.

friendly daemon) dishes up that information (if it has it), TCP magic sends it to the client, then the client prints the information in your terminal. We will start our project with a little TCP echo server that prints the literal text the client sends so that we can understand the cases we're dealing with.

Project overview

To kick this off, we'll use Stack with the `stack new` command like so:

```
$ stack new fingerd simple
```

This gets us a simple project with a single executable stanza in the Cabal file. The final version after we've added `Debug.hs` will have the following layout:

```
$ tree .
.
├── LICENSE
├── Setup.hs
├── fingerd.cabal
├── src
│   ├── Debug.hs
│   └── Main.hs
└── stack.yaml
```

fingerd.cabal

Our Cabal file will mention an executable we're not going to give you yet, so you can leave the placeholder Stack generated there for now. Note that we have gently reformatted the text to fit this book's format:

```
name:         fingerd
version:      0.1.0.0
synopsis:     Simple project template
description:  Please see README.md
homepage:     https://github.com/u/fingerd
license:      BSD3
license-file: LICENSE
```

```
author:       Chris Allen
maintainer:   cma@bitemyapp.com
copyright:    2016, Chris Allen
category:     Web
build-type:   Simple
cabal-version: >=1.10

executable debug
  ghc-options:         -Wall
  hs-source-dirs:      src
  main-is:             Debug.hs
  default-language:    Haskell2010
  build-depends:       base >= 4.7 && < 5
                     , network

executable fingerd
  ghc-options:         -Wall
  hs-source-dirs:      src
  main-is:             Main.hs
  default-language:    Haskell2010
  build-depends:       base >= 4.7 && < 5
                     , bytestring
                     , network
                     , raw-strings-qq
                     , sqlite-simple
                     , text
```

Now that we have taken care of that, let's write some code.

src/Debug.hs

This is our first source file. We're going to use this program to show what the client sends us and then send it back. In this respect, it's almost identical to the echo server demonstrated in the documentation of the `network`[3] library we're relying on. The difference is that it also prints a literal representation of the text that is sent.

Our debug program is a TCP server, similar to a web server that provides a web page but lower-level and limited to sending raw text

[3] `https://www.stackage.org/package/network`. The example we're referring to is in the `Network.Socket.ByteString` module. Click on it, and look for the example.

back and forth. What is different is that a web server communicates with browsers over a TCP socket using a structured protocol rich with metadata, routes, and a standard describing that protocol. What we're doing is older and more primitive:

```
module Main where

import Control.Monad (forever)
import Network.Socket hiding (recv)
import Network.Socket.ByteString
       (recv, sendAll)


logAndEcho :: Socket -> IO ()
logAndEcho sock = forever $ do
  (soc, _) <- accept sock
  printAndKickback soc
  sClose soc

  where printAndKickback conn = do
          msg <- recv conn 1024
          print msg
          sendAll conn msg
```

This sets up our server. Its argument is a socket (`sock`) that listens for new client connections; due to our use of `forever`, that socket remains open indefinitely. The `accept` action will block until a client connects to the server. The socket `soc` is the result of `accept`-ing a connection for communicating with the client.

The server can receive up to 1024 bytes of text from the client. All it does here is print the text literally, then echo what the client sends right back to the client that made the connection. Note that `recv` is permitted to return fewer than the maximum bytes specified if that's all the client sends. Then the connection to the client is closed—we apply `sClose` to `soc` but not to `sock`, so `sock`, the server socket, remains open. Because this action loops forever, the next thing we do is await another client connection:

```
main :: IO ()
main = withSocketsDo $ do
```

```
  addrinfos <- getAddrInfo
               (Just (defaultHints
                 {addrFlags =
                    [AI_PASSIVE]}))
               Nothing (Just "79")

  let serveraddr = head addrinfos
  sock <- socket (addrFamily serveraddr)
                 Stream defaultProtocol

  bindSocket sock (addrAddress serveraddr)
  listen sock 1
  logAndEcho sock
  sClose sock
```

At the beginning of `main`, `withSocketsDo` is not going to do anything at all unless you're on Windows. If you *are* on Windows, it's obligatory to use the sockets API in the `network` library. The address information stuff is mostly noise and can be ignored as a means for describing what kind of TCP server we're firing up and what port it's listening on.

The important part is the `(Just "79")` part—that's the port we're listening for connections on. Also note that you'll need administrative privileges on most operating systems to listen on that port.

TCP socket libraries like `network` often call everything a socket. Server listening for connections? That's a socket. Client connection that you were listening for? That's a socket. Everything's a socket, and nothing's a wrench.

The next bit constructs a sort of socket descriptor with `socket`. Then, we bind the socket to the address (port) we want. Lastly, we let the operating system know we're prepared to listen for connections from clients with `listen`. From there, we fire off our server logic, which runs indefinitely. If and when `logAndEcho` finishes, we'll close the socket server, and then our story is over.

The next step, assuming your project is built, is to fire up the debug server—note that it'll want administrative privileges for using port 79:

```
$ sudo `stack exec which debug`
```

```
...build noise and a password prompt...
```

That will get our echo server set up, and we can now test it using `telnet` to connect. Telnet is often used to debug TCP services that use text to communicate. Note that you'll need to use `sudo` or otherwise make use of administrator powers to start the program, because it wants to use a network port that only administrators or root accounts have access to in most operating systems. Usually, this is the first 1024 ports. Once you have the debug server running in one terminal, you'll connect to it from a new terminal, like so:

```
$ telnet localhost 79
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
```

From there, `telnet` is waiting for you to type something and then hit enter:

```
blah
blah
Connection closed by foreign host.
```

In the above, we type `blah`, hit enter, and get `blah` echoed back to us. Then, the server closes the connection. Remember that `sClose` is applied to `soc` in our `logAndEcho` function, ensuring that the temporary `telnet` connection is closed. However, the server is still open, and you can make further requests by reopening the connection.

Let us take a look at the server side to see what it prints:

```
"blah\n"
```

We use `print` rather than `putStrLn` in `logAndEcho` on purpose, so we can get a literal representation of the data that is sent. In this case, the string `"blah"` followed by the special character `"\n"` is sent. On Unix-based operating systems such as Linux, `"\n"` is the default line-ending character. Microsoft Windows uses `"\r"` followed by `"\n"` for the same purpose.

Having done that, let us now do the same thing with a finger client:

```
$ finger callen@localhost
[localhost]
Trying 127.0.0.1...
callen
$ finger @localhost
[localhost]
Trying 127.0.0.1...
```

Particularly if you're on a Mac, you may get some noise here that looks like this:

```
Trying ::1...
finger: connect: Connection refused
Trying 127.0.0.1...
```

It should connect after that. It attempts to use IPv6 first to reach your finger daemon; when it can't, it should use IPv4. You can probably ignore this.

Then the output server-side for this would be:

```
"callen\r\n"
"\r\n"
```

The first command asks the finger daemon running at `localhost` for information on the user `callen`; the second asks for a listing of users. With the printed output the server gives us, we now know what queries from a finger client will look like to our TCP server. With that done, we'll now write up the final TCP server itself.

31.4 Slightly modernized fingerd

Historically, the data that `finger` returns about users was part of the operating system. That information is still typically stored in the OS, but for security reasons, it's no longer routinely shared through finger requests. We're going to update the source of data for `finger` by using an embedded SQL database called SQLite. A database is a convenient yet robust way of sorting and reading data, and SQLite is a lightweight database. The data will be stored in a file within the

main project directory, so there won't be a lot of mystery or magic involved in interacting with it.

First, we'll show you the TCP server's framing of the logic, then we'll show you how the database interaction works. From here, all the code goes into your `Main.hs` file:

```
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE QuasiQuotes       #-}
{-# LANGUAGE RecordWildCards   #-}
```

`OverloadedStrings` you already know. `QuasiQuotes` is for the literals, which you've seen before. `RecordWildCards` is the new one and isn't too difficult to figure out. It spares us manually yanking the contents of a record into scope; instead, the record accessors become bindings to the contents such that, given the following code:

```
{-# LANGUAGE RecordWildCards #-}

module RWCDemo where

data Blah =
  Blah { myThing :: Int }

wew Blah{..} = print myThing
```

`wew` will print the `myThing` inside of the `Blah` argument it is applied to without needing to apply `myThing` to a `Blah` value or to destructure the contents of `Blah` in the pattern match. It's purely a convenience.

Next, we'll need the ability to decode a `Text` value from a UTF-8 `ByteString` and then re-encode a `Text` value as a UTF-8 `ByteString`:

```
module Main where

import Control.Exception
import Control.Monad (forever)
import Data.List (intersperse)

import Data.Text (Text)
import qualified Data.Text as T
import Data.Text.Encoding
       (decodeUtf8, encodeUtf8)
```

```
import Data.Typeable
import Database.SQLite.Simple
       hiding (close)
import qualified Database.SQLite.Simple
       as SQLite

import Database.SQLite.Simple.Types
import Network.Socket hiding (close, recv)
import Data.ByteString (ByteString)

import qualified Data.ByteString as BS
import Network.Socket.ByteString
       (recv, sendAll)
import Text.RawString.QQ
```

Creating the database We're using the `sqlite-simple` library to make a self-contained database stored in a file in the same directory as our project. This will act as the repository of users our finger daemon can report on.

Now, we dig into where the data comes from. `User` is the datatype describing our user records:

```
data User =
  User {
      userId   :: Integer
    , username :: Text
    , shell :: Text

    , homeDirectory :: Text
    , realName :: Text
    , phone :: Text
  } deriving (Eq, Show)
```

It's not super-structured or interesting, but it gets things rolling. The only bit potentially out of the ordinary here is that we have a `userId` field of type `Integer` in order to provide the database with what's called a *primary key*. This is a means of uniquely identifying data in the database independently of the text fields in our record type, among other things.

We need some boilerplate type class instances for marshalling and unmarshalling data to and from the SQLite database:

```
instance FromRow User where
  fromRow = User <$> field
                 <*> field
                 <*> field
                 <*> field
                 <*> field
                 <*> field

instance ToRow User where
  toRow (User id_ username shell homeDir
              realName phone) =
    toRow (id_, username, shell, homeDir,
           realName, phone)
```

This should remind you of `FromJSON` and `ToJSON`.

```
createUsers :: Query
createUsers = [r|
CREATE TABLE IF NOT EXISTS users
  (id INTEGER PRIMARY KEY AUTOINCREMENT,
   username TEXT UNIQUE,
   shell TEXT, homeDirectory TEXT,
   realName TEXT, phone TEXT)
|]
```

The `Query` type is a newtype wrapper for a `Text` value. Conveniently, `Query` has an `IsString` instance, so string literals can be `Query` values. This isn't *really* a query, though; it's a SQL statement defining the database table that will contain our user data. The primary key stuff is noise saying that the row is named `id` and that we want that field to auto-increment without needing to do it ourselves. That is, if the last row we insert into the database has the id `1`, then the new one will be auto-assigned the primary key `2`. The rest of it describes field names and their representation (`TEXT`), but you'll note we require usernames to be unique, so that there cannot be two `User` values with the same username.

```
insertUser :: Query
insertUser =
  "INSERT INTO users\
  \ VALUES (?, ?, ?, ?, ?, ?)"

allUsers :: Query
allUsers =
  "SELECT * from users"

getUserQuery :: Query
getUserQuery =
  "SELECT * from users where username = ?"
```

This is utility stuff for inserting a new user, getting all users from the user table, and getting all the fields for a single user with a particular username. The question marks are how the `sqlite-simple` library parameterizes database queries.

```
data DuplicateData =
  DuplicateData
  deriving (Eq, Show, Typeable)

instance Exception DuplicateData
```

The type above is a one-off exception we throw whenever we get something other than zero or one users for a particular username. That should be impossible, but you never know.

```
type UserRow =
  (Null, Text, Text, Text, Text, Text)
```

`UserRow` is a type synonym for the tuples we insert to create a new user.

```
getUser :: Connection
        -> Text
        -> IO (Maybe User)
getUser conn username = do
  results <-
    query conn getUserQuery (Only username)
```

```
  case results of
    [] -> return $ Nothing
    [user] -> return $ Just user
    _ -> throwIO DuplicateData
```

The `Only` data constructor is how we pass a single argument instead of a 2-or-greater tuple to our query parameters when using the `sqlite-simple` library. This is needed, because `base` has no 1-tuple type and `getUserQuery` takes a single parameter. We check for none, one, or many results and convert it into a `Nothing`, `Just`, or `IO` exception.

Finally, we need a utility function for creating the database with a single example row of data:

```
createDatabase :: IO ()
createDatabase = do
  conn <- open "finger.db"
  execute_ conn createUsers
  execute conn insertUser meRow

  rows <- query_ conn allUsers
  mapM_ print (rows :: [User])
  SQLite.close conn

  where meRow :: UserRow
        meRow =
          (Null, "callen", "/bin/zsh",
           "/home/callen", "Chris Allen",
           "555-123-4567")
```

Stack may balk, because you have a module called `Main` that has no `main` defined. If that's the case for you, you can do this:

```
main :: IO ()
main = createDatabase
```

We'll change that `main` later, but this will get your executable building for now.

Running this a second time will result in an error without changing the database. If you need or want to reset the database, you can delete the `finger.db` file.

Before you continue The code that follows will assume and require that a SQLite database with the name `finger.db` and with the schema outlined in `createUsers` exists in the same directory where you run your `fingerd` service.

To run `createDatabase`, you could do the following:

```
$ stack ghci --main-is fingerd:exe:fingerd
...noise noise...
Prelude> createDatabase
User {userId = 1, ... noise ... }
```

With that in place, you can continue implementing your finger daemon.

Let your fingers do the walking

We're still in our `Main` module here. You should have created the database already, but now we'll write the functions that will allow the server to listen and respond to client queries:

```
returnUsers :: Connection
            -> Socket
            -> IO ()
returnUsers dbConn soc = do
  rows <- query_ dbConn allUsers

  let usernames = map username rows
      newlineSeparated =
        T.concat $
        intersperse "\n" usernames

  sendAll soc (encodeUtf8 newlineSeparated)
```

`returnUsers` uses a database `Connection` and a `Socket` for talking to the user. The database connection is used to get a list of all the users in the database, which is then changed into a newline-separated `Text` value. Then, that is encoded into a UTF-8 `ByteString`, which is sent through the socket to the client.

```
formatUser :: User -> ByteString
formatUser (User _ username shell
            homeDir realName _) = BS.concat

  ["Login: ", e username, "\t\t\t\t",
   "Name: ", e realName, "\n",
   "Directory: ", e homeDir, "\t\t\t",
   "Shell: ", e shell, "\n"]
  where e = encodeUtf8
```

This function is used to format `User` records as a UTF-8 `ByteString` value. The format is intended to mimic popular `fingerd` implementations, but we're not going for precision here.

```
returnUser :: Connection
           -> Socket
           -> Text
           -> IO ()
returnUser dbConn soc username = do
  maybeUser <-
    getUser dbConn (T.strip username)

  case maybeUser of
    Nothing -> do
      putStrLn
        ("Couldn't find matching user\
          \ for username: "
        ++ (show username))
      return ()

    Just user ->
      sendAll soc (formatUser user)
```

This is the single user query case, where we use `formatUser` to provide detailed information to the client on a single user. We have to handle the case where no user by the username provided is found. As it stands, the `Nothing` case here will print the report that no user was found by that username in the server terminal but will not send that information—or any information—to the client side. You may

want to change that, as it might be useful to tell the end user why no information was returned.

If a user is found, we send the formatted `ByteString` of the `User` record to the client. The stripping of the username text prior to querying is because the literal data sent for a username query is `"yourname\r\n"`, and in order for that to match `"yourname"`, we need to strip the control characters from the text, which `strip` from `Data.Text` does for us.

```
handleQuery :: Connection
            -> Socket
            -> IO ()
handleQuery dbConn soc = do
  msg <- recv soc 1024

  case msg of
    "\r\n" -> returnUsers dbConn soc
    name ->
      returnUser dbConn soc
      (decodeUtf8 name)
```

`handleQuery` receives up to 1024 bytes of data. Based on the data the client sends to the server, the case expression discriminates between when it should send a list of all users or only a single user. Fortunately, the protocol is relatively uncomplicated, so we don't have to do any parsing, as would ordinarily be required for communicating with a more elaborate protocol.

```
handleQueries :: Connection
              -> Socket
              -> IO ()
handleQueries dbConn sock = forever $ do

  (soc, _) <- accept sock
  putStrLn "Got connection, handling query"

  handleQuery dbConn soc
  sClose soc
```

It's similar to the echo server, save for the additional argument of the database connection and the logging of when connections are accepted.

Now, we need to change `main` to assemble our whole program:

```
main :: IO ()
main = withSocketsDo $ do
  addrinfos <-
    getAddrInfo
    (Just (defaultHints
      {addrFlags = [AI_PASSIVE]}))
    Nothing (Just "79")

  let serveraddr = head addrinfos
  sock <- socket (addrFamily serveraddr)
          Stream defaultProtocol
  bindSocket sock (addrAddress serveraddr)

  listen sock 1
  -- only one connection open at a time
  conn <- open "finger.db"
  handleQueries conn sock

  SQLite.close conn
  sClose sock
```

The only new bit above is the opening of a connection to a SQLite database located in the same directory as your project. The connection to the database is passed to the query-handling code, which runs indefinitely like the echo-and-log server. If it somehow stops without throwing an exception, we close the server socket, just to be good little programmers.

Now, we're done. Assuming you created a SQLite database using `createDatabase` that is valid and accessible to your program, we can finally run our program. You'll want to do this in one terminal window, on its own:

```
$ stack build
$ sudo `stack exec which fingerd`
```

Then, in another, different shell session, the following should work:

```
$ finger callen@localhost
Login: callen             Name: Chris Allen
Directory: /home/callen   Shell: /bin/zsh
```

And that's it. In the exercises, we've suggested some ways to extend this application, and we hope you've enjoyed this little foray into TCP sockets and basic networking. Security concerns aside, the finger protocol has been used over the years for some pretty cool things. Perhaps most famously, John Carmack used `.plan` files as a kind of microblog to deliver updates on the development process of Quake.[4]

31.5 Chapter exercises

1. Try using the `sqlite3` command line interface to add a new user or modify an existing user in `finger.db`.
2. Write an executable separate from `fingerd` and `debug` that allows you to add new users to the database.
3. Add the ability to modify an existing user in the database.
4. Try creating a "control socket" bound to a different port that permits inserting new data into the database while the server is running. This will probably require, at minimum, learning how to use `forkIO` and the basics of concurrency in Haskell, among other things. Design the format for representing the user rows passed over the TCP socket yourself. For bonus points, write your own client executable that accepts arguments from the command line, as well.
5. Celebrate completing this massive book.

[4] `https://github.com/bitemyapp/john-carmack-plan-archive`

Index