Haskell Programming
from first principles

Christopher Allen

Julie Moronuki

Pure functional programming without fear or frustration

Contents

License

Concerning Code

You may use and redistribute example code from this book for non-commercial purposes if you acknowledge their source and authorship. The origins of the code should be noted in any documentation attached to where the code was used as well as in the source code itself (as a comment). The attribution should include "Haskell Programming from first principles" and the names of the authors, Chris Allen and Julie Moronuki.

All to say, fair use applies. If you're incorporating the content of the book into anything that derives revenue, you need to talk to us first.

Authors' preface

Chris's story

I've been programming for over 15 years, 8 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 devs (including myself) couldn't resolve in less than 2 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've had similar happen in Python and Common Lisp as well. 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. The same issue in Haskell would be trivially resolved in a minute or less because the type-checker will identify precisely where you were 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 that 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 pedagogy has made it hard for many people to learn.

It doesn't have to be that way.

I've spent the last two years actively teaching Haskell online and in person. 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.

Julie's story

I met Chris Allen in spring 2014. We met on Twitter and quickly became friends. As anyone who has encountered Chris–probably in any medium, but certainly on Twitter–knows, it doesn't take long before he starts urging you to learn Haskell.

I told him I had no interest in programming. I told him nothing and nobody had ever been able to interest me in programming before. When Chris learned of my background in linguistics, he thought I might be interested in natural language processing and exhorted me to learn Haskell for that purpose. I remained unconvinced.

Then he tried a different approach. He was spending a lot of time gathering and evaluating resources for teaching Haskell and refining his pedagogical techniques, and he convinced me to try to learn Haskell so that he could gain the experience of teaching a code-neophyte. Finally, with an "anything for science" attitude, I gave in.

Chris had already known that the available Haskell learning materials each had problems, but I don't think even he realized just how frustrating they would be to me. All of the materials I ran across relied on a background with other programming languages and left many terms undefined or explained features of Haskell by analogy (often faulty) to features of other languages–features I had no experience with.

When I say I had no programming experience, I really, truly mean it. I had to start from learning what a compiler does, what version control means, what constitutes side effects, what is a library, what is a module, what on earth is a stack overflow. At the time of this writing, that is where I was less than a year ago; by the time we finish writing this book and it is published, it will be almost two years.

I didn't think I would stick with it. It was too frustrating, too time-consuming, and I saw no practical purpose in it anyway. I felt like I wouldn't be able to learn it. Working with Chris kept me motivated and interested, though.

Eventually, as he realized that a new type of book for learning Haskell was necessary, he decided to write one. I agreed at the time to be his guinea pig. He would send me chapters and I would learn Haskell from them and send feedback. Through the fall, we worked like this, on and off, in short bursts. Eventually we found it more efficient for me to take on authorship duties. We developed a writing process where Chris made the first pass at a chapter, scaffolding the material it needed to cover. Then I filled in the parts that I understood and came up with questions that would elaborate and clarify the parts I didn't already know. He answered my questions until I understood, and I continued adding to and refining what was there. We each wrote exercises–I write much easier ones than he does, but the variety is beneficial.

I have tried, throughout the process, to keep thinking from the perspective of the absolute beginner. For one thing, I wanted my own understanding of Haskell to deepen as I wrote so I kept questioning the things I thought I knew. Also, I wanted this book to be accessible to everyone.

In interacting with other Haskell learners I often hear that other materials leave them feeling like Haskell is difficult and mysterious, a programming language best left to wizards.

It doesn't have to be that way.

Haskevangelism

The rest of this preface will give some background of Haskell and will make reference to other programming languages and styles. If you're a new programmer, it is possible not all of this will make sense, and that's okay. The rest of the book is written with beginners in mind, and the features we're outlining will make more sense as you work through the book.

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

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

1. User defined record types

2. User defined sum types (unions not limited to simple enumerations)

3. Switch/case expressions which support the sum types

4. Compile-time enforced constant values, declared with `=` rather than `:=`

5. Unified syntax for using value and reference types. No manual pointer dereferencing

6. Closures with lexical scoping (without this, many functional patterns fall apart)

7. Implementation-agnostic parallelized execution of procedures

8. Multi-pass compilation - you can declare stuff after you use it.

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

Languages like Java *just* acquired lambdas and lexical-scoping as of version **8**. Haskell leverages more of the developments in programming languages invented since ALGOL68 than most languages in popular use, but with the added benefit of a mature implementation and sound design. Sometimes we hear Haskell being dismissed as "academic" because it is relatively up-to-date with the current state of mathematics and computer science research. In our view, that progress is good and helps us solve practical problems in modern computing and software design.

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

There are other ways to prevent scurvy, just as there are other programming languages you can use to write software. Or perhaps you are an explorer who doesn't believe scurvy can happen to you. But packing lemons provides some insurance on those long voyages. Similarly, having Haskell in your toolkit, even when it's not your only tool, provides type safety and

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

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

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 have sent them each some very rough material, and they have been willing to work with it and send detailed feedback about what worked and what didn't work. Their reviews helped us ensure the book is both suitable for beginners and comprehensive. Also, they're both just wonderful people all around.

Martin DeMello and Daniel Gee have been steadily working their way through the entire book, doing a full review on all the material, and offering smart criticisms and helpful suggestions. They have given us a lot to consider as we go back to do revisions for the final draft.

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 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 very pleased and grateful to have their input. The book would be messier and the exercises less useful if not for their help. Julien Baley has been incredible on this front, not only representing a non-trivial portion of our reader feedback over the course of several later releases of the book, but also catching things nobody else noticed.

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.

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 Haskell

If you are new to programming entirely, Haskell is a great first language. You may have noticed the trend of "Functional Programming in [Imperative Language]" books and tutorials and learning Haskell gets right to the heart of what functional programming is. Languages such as Java are gradually adopting functional concepts, but most such languages were not designed to be functional languages. We would not encourage you to learn Haskell as an *only* language, but 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.

If you are already a programmer, writing Haskell code 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. Without wanting to trivialize the learning curve, we would argue that part of being a professional means accepting that your field will change as new information and research demands it.

It also means confronting human frailty. In software this means that we *cannot* track all relevant metadata about our programs in our heads ourselves. We have limited space for our working memory, and using it up for

anything a computer can do for us is counter-productive. We don't write Haskell because we're geniuses — we use tools like Haskell because we're not geniuses and it helps 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.

What is Haskell for?

Haskell is a general purpose, functional programming language. It's not scoped to a particular problem set. 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.

Haskell's ecosystem has an array of well-developed libraries and tools. Hoogle is a search tool that allows you to search for the function you want by type signature. It has a sophisticated structural understanding of types and can match on more than just syntax. Libraries such as Yesod and Scotty allow you to build web applications quickly, each addressing a different niche of web development. Aeson is popular and in wide use in the Haskell community for processing JSON data which is currently the lingua franca for data serialization and transmission on the web. Gloss is popular for rendering 2d vector graphics. Xmonad is a tiled window manager for Linux/X11, written in Haskell and popular with Haskell users and non-Haskellers. Many more people use pandoc which is as close to a universal document conversion and processing tool as currently exists, and is also written in Haskell.

Haskell is used in industrial settings like oil & gas control systems, data anonymization, mobile applications for iOS and Android, embedded software development, REST APIs, data processing at scale, and desktop applications. Entire financial front offices have used Haskell to replace everything from C++ trading applications to apps running in Excel, and even some indie game developers are using Haskell with functional reactive programming

libraries to make their projects.

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

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

If you are an experienced programmer, learning Haskell is more like learning to program all over again. Getting somebody from Python to JavaScript is comparatively trivial. They share very similar semantics, structure, type system (none, for all intents and purposes), and problem-solving patterns (e.g., representing everything as maps, loops, etc.). Haskell, for most who are already comfortable with an imperative or untyped programming language, will impose previously unknown problem-solving processes on the learner. 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 you are learning Haskell as a first language, you may have experienced a specific problem with the existing Haskell materials and explanations: they assume a certain level of background with programming, so they frequently explain Haskell concepts in terms, by analogy or by contrast, of programming concepts from other languages. This is obviously 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 the Haskell concepts, 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.

This book is not something you want to just leaf through the first time you read it. It 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 all of 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*. We do not recommend that you pass from one section of the book to the next without doing at least a few of the exercises. If you do get to a later chapter and find you did not understand a concept or structure well enough, you can always return to an earlier chapter and do more exercises until you understand it. The Freenode IRC channel `#haskell-beginners` has teachers who will be glad to help you as well, and they especially welcome questions regarding specific problems that you are trying to solve.[1]

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 the functional style of programming.

The exercises in the first few chapters are designed to rapidly familiarize you with basic Haskell syntax and type signatures, but you should expect exercises to grow more challenging in each successive chapter. Where possible, reason through the code samples and exercises in your head first, then type them out — either into the REPL[2] or into a source file — and check to see if you were right. This will maximize your ability to understand and reason 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.

---

[1]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 if you're interested in getting to know the community in their natural habitat.

[2]A REPL, short for Read-Eval-Print-Loop, is a program you use to type code in and see how the expressions evaluate. We'll explain this when we introduce GHCi.

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

What's in this book?

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 typesafe functional code but to understand it on a deep enough level that you can go from here to more advanced Haskell projects in a variety of ways, depending on your own interests and priorities.

One of the most common responses we hear from people who have been trying to learn Haskell is, "OK, I've worked through this book, but I still don't know how to actually write anything of my own." We've combined practical exercises with principled explanations in order to alleviate that problem.

Each chapter focuses on different aspects of a particular topic. We start with a short introduction to the lambda calculus. What does this have to do with programming? All modern functional languages are based on the lambda calculus, and a passing familiarity with it will help you down the road with Haskell. If you've understood the lambda calculus, understanding the feature of Haskell 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 glance at those chapters, decide they are too easy and skip them. *Please do not do this.* Even if those first chapters are covering concepts you're familiar with, it's important to spend time getting comfortable with Haskell's rather terse syntax, making sure you understand the difference between working in the REPL and working in source files, and becoming familiar with the compiler's sometimes quirky error messages. Certainly you may

work quickly through those chapters — just don't skip them.

From there, we build both outward and upward so that your understanding of Haskell both broadens and deepens. When you finish this book, you will not just know what monads are, you will know how to use them effectively in your own programs and understand the underlying algebra involved. We promise — you will. We only ask that you do not go on to write a monad tutorial on your blog that explains how monads are really just like jalapeno poppers.

In each chapter you can expect:

- additions to your vocabulary of standard functions;

- syntactic patterns that build on each other;

- theoretical foundations so you understand how Haskell works;

- illustrative examples of how to read Haskell code;

- step-by-step demonstrations of how to write your own functions;

- explanations of how to read common error messages and how to avoid those errors;

- exercises of varying difficulty sprinkled throughout;

- definitions of important terms.

We have put the definitions at the end of each chapter. 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.

A few words about working environments

This book does not offer much instruction on using the terminal and text editor. The instructions provided assume you know how to find your way

around your terminal and understand how to do simple tasks like make a directory or open a file. Due to the number of text editors available, we do not provide specific instructions for any of them.

If you are new to programming and don't know what to use, consider using an editor that is unobtrusive, uses the Common User Access bindings most computer users are accustomed to, and doesn't force you to learn much that is new (assuming you're already used to using commands like ctrl-c to copy), such as Gedit or Sublime Text. When you're initially learning to program, it's better to focus on learning Haskell rather than struggling with a new text editor at the same time. However, if you believe programming will be a long-term occupation or preoccupation of yours, you may want to move to text editors that can grow with you over time such as Emacs or Vim.

A few words about the examples and exercises

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

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

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

- You are intended to type *all* of the examples into the REPL or a file and load them. We *strongly* encourage you to attempt to modify the example and play with the code after you've made it work. Forming hypotheses about what effect changes will have and verifying them is critical!

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

- Not every example is designed to be entered into the REPL. Not every example is designed to be entered into a file. We explain the syntactic differences between files and REPL expressions. You are expected to perform the translation between the two yourself after it has been explained. This is so you are accustomed to working with code in an interactive manner by the time you finish the book.

- Exercises in the body of the chapter are usually targeted to the information that was covered in the immediately preceding section(s).

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

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

- We ask you to write and then rewrite (using different syntax) a lot of functions. Few problems have only one possible solution, and solving the same problem in different ways increases your fluency and comfort with the way Haskell works (its syntax, its semantics, and in some cases, its evaluation order).

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

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

- It is better to type the code examples and exercises yourself rather than copy and paste. Typing it out yourself makes you pay more attention to it, so you ultimately learn much more from doing so.

- We recommend you move away from typing code examples and exercises directly into GHCi sooner rather than later and develop the habit of working in source files. It isn't just because the syntax can become unwieldy in the REPL. 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 the REPL from the file to run it, and we'll cover how to do this in the appropriate chapter.

- Another benefit to writing code in a source file and then loading it into the REPL is that you can write comments about the process you went through in solving a problem. Writing out your own thought process

can clarify your thoughts and make the solving of similar problems easier. At the very least, you can refer back to your comments and learn from yourself.

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

```
f = undefined
```

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

Chapter 1

All You Need is Lambda

Anything from almost nothing

> 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. The lambda calculus was devised in the 1930s by Alonzo Church as a way to formalize the concept of effective computability, serving the same purpose as the Turing machine but de-emphasizing the "machine" part. The goal of formalizing a model of computation is to determine which problems, or classes of problems, can be solved.

Right now you may be asking yourself, "what happened, where's the Haskell, how did I get here?" You may be contemplating skipping this chapter as irrelevant to practical programming. You may feel tempted to skip ahead to the "fun stuff" when we build a project.

DON'T.

We know from prior experience that this is the leading way students of Haskell burn out. They skip the parts that seem boring and simple and try to rush into building a project. They want to cargo cult the stuff they don't understand just so they can *do* something. Then, at some point, they realize they don't understand, don't know where things went wrong, and run away from Haskell with the impression that Haskell is hard and scary.

That pattern is unnecessary, and we don't want you to burn out. That just ain't how we roll here. So 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. Here is your foundation.

Why learn the lambda calculus?

Because Haskell is a pure functional programming language.

So let's talk about what it means to be a pure functional programming language.

No. Let's back up one step further and talk about functions.

1.2 What's a function?

You're probably here to learn "functional programming" so it's worth learning what a function, all by itself and outside the context of programming languages is. A function is a relation between two sets, a set of inputs and a set of outputs. The function itself defines and represents the relationship. The set of inputs is referred to as the domain and the set of outputs the codomain. Here a set is a collection of unique values. The set $\{1, 2, 3\}$ is perfectly fine, whereas $\{1, 1, 2\}$ is not because it has duplicate values. This uniqueness property is important for reasons we'll explain in a bit.

Thus, if we 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
```

Our domain (inputs) is the set $\{1, 2, 3\}$ and the codomain (outputs) is $\{A, B, C\}$. A crucial point about how these relations are defined: our hypothetical function will *always* return the value A given the input 1 — no exceptions! The following is emphatically *not* a valid function:

```
f: 1 -> X
f: 1 -> Y
f: 2 -> Z
```

You cannot return two different results for the same input; the notion doesn't even make sense. This accords nicely with our understanding of the uniqueness of values in sets. Just as a given value can only occur once in a set, a function can only relate a value in its domain to a single unique value in the codomain.

$f(x) = x + 1$

$f(1) = 1 + 1$

$f(1) = 2$

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

What is functional programming?

Functional programming is a computer programming paradigm that relies on definition and application of functions as the basis for programs (contrast this to the sequential instructions you find in imperative programs). Functions here are modeled on the mathematical functions described above.

The essence of functional programming is that programs are a combination of expressions to be evaluated. Expressions are a superset that includes concrete values, variables, and functions. Functions have a more specific definition: they are expressions that are applied to an argument and, once applied, can be reduced or evaluated. In Haskell, and in FP more generally, functions are first-class: they can be used as values or arguments to yet more functions.

Functional programming is not new, but it is undergoing something of a renaissance, and Haskell is at the leading edge of FP. The resurgence of FP has been attributed to several factors, among them the need for better methods of handling parallel and concurrent processing and the ease of maintaining and refactoring large codebases.

Perhaps it is also that functional programming is based on simple, powerful mathematical ideas, and math itself never gets old.

About purity

Functional programming languages are all based on the lambda calculus and rely on expressions, but they are not all equally pure. Pure FP is just the lambda calculus. Impure FP usually adds ambient effects, mutation, and imperative step-wise encoding of instructions. Functions are represented as lambda terms in the lambda calculus.

If you're new to programming and unfamiliar with mutation and side-effects,

this can be a difficult concept to grasp, but it's not as arcane as it seems. The main point here is 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. The benefits of writing programs this way include ease of reasoning about programs, debugging, refactoring, and maintaining code and increased ease of evaluating expressions in parallel. Haskell does have ways to use effects for all the same purposes as impure languages. However, Haskell does this in a way such that the effects are proper values like everything else in the language, explicit, and functional purity (still just lambdas) is maintained.

Being pure and functional also leads to Haskell having a high degree of abstraction and composability. Abstraction allows you to write shorter, more concise programs by factoring out common, repeated structures into more generic code that can be reused with different values and data structures. Haskell programs are built up as separate, independent functions. Some people liken it to building with Legos: the functions are bricks that can be assembled and reassembled as needed.

These features also make Haskell's syntax rather minimalist. Function calls, for example, are just white space. This can be jarring, especially for experienced programmers, but you might come to appreciate the saved keystrokes once you get used to it.

Minimalist syntax, abstract, and pure computations.

Just like the lambda calculus.

1.3 Lotsa lambdas

The lambda calculus has three basic components: expressions, variables, and abstractions. An expression can be a variable name, an abstraction (that is, a lambda itself), a function application (a variable or lambda applied to an argument), or some combination of all these. The simplest expression is a single variable. The variable has no meaning or value. It just gives a name to a potential input to a function.

Functions, also called abstractions, consist of two parts: the head and the

body. Functions are a way of describing an operation that, given an input, produces a specific output. The head of the function is a λ (lambda) followed by a variable name that is the parameter or argument of that function. The body of the function is another expression. So, a simple function might look like this:

```
λ x . x
```

This is a lambda abstraction of x. 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. Because it has no name, we call it an anonymous function. But where our f function could be called by name, for example in another function, the lambda abstraction cannot.

Lambda abstractions have the following basic structure:

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

λ x . x
  ^——————— the single argument the lambda takes.
          the variable is bound when the function
          is applied.

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

The . separates the arguments of the lambda from the function body. The only argument it introduces is x. This also happens to be the identity function in the lambda calculus, so all it does is accept a single argument x and return that same argument. Here x is bound by the enclosing lambda because it is defined when the lambda is applied to an 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 be bound to the same value.

Let's allow ourselves a moment to bring this into the real world and apply this function to, say, a number. In fact, the lambda calculus derives numbers from lambda abstractions as well, rather than using the numerals we are familiar with, but the applications can become quite cumbersome and difficult to read, so we're going to use numerals to demonstrate the evaluation of the identity function above.

When we apply the function to an argument, we simply eliminate the head of the function (because it has done its job, which was only to bind a variable) and replace the instances of bound variables with the argument. In this case, since there is only one variable, the process is not terribly interesting:

```
λ x . x (2)
-- eliminate the head
-- substitute 2 for each x in the body

2
```

Huzzah! The identity of 2 is 2! All right, we admitted it wasn't terribly interesting, but now you have learned the basic rule of computation in the lambda calculus.

1.4   Beta reduction

As we saw above, function application consists of applying the abstraction to another expression. When you apply the function, you substitute the input expression for all instances of bound variables within the abstraction, a process known as beta-reduction. At the same time you eliminate the head that binds the variables. When there are no more heads, or lambdas, left, you're done. A computation therefore consists of an initial lambda expression (or two, if you want to separate the function and its input) plus

a finite sequence of lambda terms, each deduced from the preceding term by one application of beta-reduction.

The function $\lambda x.xy$ might be applied to an argument, z like this: $(\lambda x.xy)z$. We would bind the argument x to z and replace any occurrence of the bound variable x in the body of the function with z to see the result. We'd be left with zy which is "z applied to y". This means that z is a function expecting an argument, but we can't apply (or reduce) it until we know the terms of z. This can't be reduced any further because we don't know (or need to know) the terms of z or y and nothing remains to be applied.

1. $\lambda x.xy$
   We're going to show you more steps than most literature on lambda calculus will bother with, so you know why everything happens. The above is the lambda we'll apply to an argument in the next step.

2. $(\lambda x.xy)z$
   We apply the lambda to the argument z. The argument x is bound to z. There are no more arguments to apply in the head so the enclosing lambda has been applied out of existence.

3. $(\lambda[x := z].xy)$
   Now we can replace all occurrences of x with z. The notation $[x := z]$ indicates that x is bound to z, so z will be substituted for all occurrences of x in the expression.

4. $(\lambda[x := z].zy)$
   Replace the x with z, thereby changing expression xy into zy.

5. zy
   Drop the head of the lambda, since it's been applied away. We can reduce this no further.

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 1:

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

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

This reduction is really all we can do in the lambda calculus. We keep following the rules of application, substituting arguments in for bound variables until there are no more heads left to evaluate or no more arguments to apply them to.

1.5 Bound and free variables

We have seen how the head of a lambda abstraction binds variables. The purpose of the head of the function is really 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:

$$\lambda x.xy$$

The x here is a bound variable because it shows up in the head of the function as well as the body. But the y is not mentioned in the head of this function, so it is a free variable. When we apply this function to an argument, nothing can be done with the y. It remains unreducible.

We can bind multiple variables in a function:

$$\lambda xy.xy$$

However, it is important to note that we have now actually created a function with multiple, nested heads. When we go through the beta-reduction, we eliminate one head at a time, starting from the outermost, or leftmost, head. More on this in just a moment. We'll demonstrate how this is the case when we talk about multiple arguments and currying.

1.6   Alpha equivalence

Ordinarily when people express the identity function in lambda calculus you'll see something like

$$\lambda x.x$$

The variable x here is customary and not semantically meaningful except in where it occurs in the expression. Because of this, there's a form of equivalence between lambda terms called *alpha equivalence*. This is a way of saying that:

$$\lambda x.x$$

$$\lambda d.d$$

$$\lambda z.z$$

All mean the same thing. They're all the identity function. Take one argument, kick it back out. This equivalence applies only to bound variables where you know for a fact within the scope of the lambda that they will be the same, even if they have different names. This intuition does not apply for free variables as they are not defined by the enclosing lambda alone.

1.7   Multiple arguments

Each lambda accepts 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 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:

$$\lambda xy.x$$

is a convenient short-hand for two nested lambdas (one for each argument x and y):

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

When you apply the first argument, you're binding x, eliminating the outer lambda, and have $\lambda y.x$ 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. $(\lambda 1.1)$

4. 1

Now with our "multiple" argument lambda:

1. $\lambda xy.x$

2. $(\lambda x.(\lambda y.x))\ 1\ 2$
   We're going to apply these two lambdas, one nested within the other.

3. $(\lambda y.1)\ 2$
   Now we apply the second lambda to the second argument. x is bound to 1 now.

4. $\lambda 2.1$
   Here we toss the head because the last lambda was applied to 2, but there are no variables in the body expression left to bind it to.

5. 1

   The final result, we did nothing with the second argument y because it didn't occur in the body anywhere.

Here applying the outer lambda which handled the argument x to 1 served to *bind* the variable x to a constant value 1. You'll see most examples working in the lambda calculus refer to abstract variables not bound to particular, concrete values. This is because the additional information is irrelevant. For your purposes, you can see the lambda calculus as being a game with a few simple rules for transforming lambdas, but no specific "meaning". We've just introduced concrete values to make the reduction somewhat easier to see.

Another example, this time one which ends in an irreducible lambda. We're going to use different variables to help make the reduction easier to read, 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've not reduced or applied anything here, just made the currying explicit.

3. $(\lambda y.\lambda z.(\lambda m.\lambda n.m)z(yz))(\lambda p.p)$
   Our first reduction step was of the *outermost* and *leftmost* lambda that *could* be reduced, which was binding the x. The lambda binding x was only reducible because it was applied to an argument. In normal order evaluation of lambda terms, if multiple terms are "outermost", we use leftmost to pick what to reduce. Here there were three outermost lambda terms, so we applied the outermost lambda of the leftmost series of lambdas that were binding x, y, and z. After we applied the outermost, leftmost lambda term "x" was bound to $\lambda m.\lambda n.m$. When we bind a variable, we can replace the occurrences of it in the terms with what it was bound to, but that doesn't mean reducing what it was replaced by until it is also the outermost and leftmost reducible term.

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 term $\lambda p.p$ it was bound to.

5. $\lambda z.(\lambda n.z)((\lambda p.p)z)$

   Now the outermost and leftmost lambda term $\lambda z.$ is irreducible because it's not being applied to anything, what remains is to go inside the terms one layer at a time until we find something reducible. In this case, the leftmost term of the next layer from the outermost was $\lambda m.\lambda n.m$ being applied to (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 $((\lambda p.p)z)$. The odd part here is that it doesn't matter what n got bound to, $\lambda n.z$ unconditionally tosses the argument and returns z.

1.8 Evaluation is simplification

There are multiple "normal forms" in lambda calculus, but here when we refer to normal form 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. In programming, we'd call this a "fully executed" program.

How do you refer to the number two? Do you say 2000 / 1000 each time or do you just say 2? The former, 2000 / 1000, is not fully evaluated because the division function has been fully applied (two arguments) and there's a simpler form it can be reduced to — the number two.

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.

Bringing this idea back over to the lambda calculus, the identity function here $\lambda x.x$ is fully reduced because it hasn't yet been applied to anything and is thus already in beta normal form. Whereas, $(\lambda x.x)z$ is *not* in beta normal 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 is just z.

1.9 Combinators

A combinator is a lambda term with no free variables. Combinators, as the name suggests, serve only to *combine* the arguments it is 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 there's one or more free variables:

1. $\lambda y.x$
   Here y is bound (it occurs 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.

The fun thing about combinators is that they can take one or more functions as their inputs and return another function as an output. Combinators provide for higher-order functions as well as certain types of recursive functions in programming. Thus combinators allow us to define complex programs with simple operations and little repetition.

We won't have a lot to say about combinators per se until later, although we will use some combinators, such as (.) and $flip$ at various times in the book, without identifying them as such.

1.10   Divergence

Not all reducible lambda terms will reduce neatly to a beta normal form. This isn't because they're already fully reduced, but rather because they *diverge.* Divergence here means that the reduction process never terminates or ends. Reducing terms should ordinarily *converge* to unique beta normal form, and divergence is the opposite of convergence. 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 $(\lambda x.xx)$ in each for each occurence of x. We're back to where we started and this reduction process never ends — we can say 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 totality more later.

1.11   Summary

Now, this is all a bit abstract. You can use the lambda calculus to compute all sorts of things, but this isn't a book about the lambda calculus.

So, what should you take away from this?

- Functional programming is based on expressions that consist of variables or constant values alone, expressions combined with other expressions, and functions.

- Expressions are evaluated, or reduced, to a result.

- Functions have a head and a body and are applied to arguments.

- 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 unique set of inputs to a unique set of outputs. Given the same input, they always return the same result.

Those things all apply to Haskell, as Haskell is a pure functional programming language, derived from the lambda calculus. As it happens, Haskell is actually based on a *typed* variation of the basic lambda calculus, but we'll save the explanation of types for the main body of the book concerning Haskell itself.

1.12   Exercises

We're going to do the following exercises a bit differently than what you'll see in the rest of the book, as we will be providing some answers and explanations for the questions below.

Combinators   Determine if each of the following 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 if each of the following 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 pencil or pen.

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

1.13   Answers

Combinators

1. $\lambda x.xxx$ is indeed a combinator, it 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, all terms are bound. The head is $\lambda xyz.$ 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 - no occurrences of free variables.

4. $\lambda xyz.xy(zxy)$ is a combinator. The lambda has the head $\lambda xyz.$ and the body: $xy(zxy)$. 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, z is free. Note that z isn't bound in the head.

Normal form or diverge?

1. $\lambda x.xxx$ doesn't diverge, has no further reduction steps. If it had been applied to itself, it *would* diverge, but by itself 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 done. This is the *omega* term we showed you earlier, just 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 are evaluated in *normal order*, which is where 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. $(\lambda x.\lambda y.xyy)(\lambda a.a)b$
   $(\lambda x.\lambda y.xyy)(\lambda a.a)b$
   $(\lambda y.(\lambda a.a)yy)(b)$
   $(\lambda a.a)(b)b$
   bb

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

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

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

6. $(\lambda a.aa)(\lambda b.ba)c$
   $(\lambda a.aa)(\lambda b.ba)c$
   $(\lambda b.ba)(\lambda b.ba)c$
   $(\lambda b.ba)(a)c$
   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) $(\lambda z1.z(a))$

   How we got there, step by step

   a) Our expression we'll reduce.

   b) Add the implied lambdas to introduce each argument.

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

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

   e) Can't apply z1 to anything, 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. $(x.z)$ gets applied to $z1$, tosses the $z1$ away and returns z. z is now being applied to $((x.a)(z1))$.

   f) Cannot reduce z further, it's free and we know nothing, so we go inside yet another nesting and reduce $((x.a)(z1))$. $x.a$ 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.

1.14   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.

   $(\lambda x.x + 1)$

   Here the head of the lambda $x.$ is abstracting out the term $x + 1$. We can apply it to any x and recompute different results for each x we applied the lambda to.

3. *Application* is how one evaluates or reduces lambdas, this binds the argument to whatever the lambda was applied to. Computations are performed in lambda calculus by applying lambdas to arguments until you run out of arguments to apply lambdas to.

   $(\lambda x.x)1$

   This example reduces to 1, the identity lambda $x.x$ was applied to the value 1, x was bound to 1, and the lambda's body is just x, so it just kicks the 1 out. In a sense, applying the lambda $x.x$ *consumed* it. We *reduced* the amount of structure we had.

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 (ie, applying or beta reducing) the leftmost outermost lambdas first, evaluating terms nested within after you've run out of arguments to apply. Normal order isn't how Haskell code is evaluated - it's *call-by-need* instead. We'll explain this more later. Answers to the evaluation exercises were written in normal order.

1.15   Follow-up resources

These are *optional* and intended only to offer suggestions on how you might deepen your understanding of the preceding topic. Ordered 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; Erik Barendsen. Introduction to Lambda Calculus
   `http://www.cse.chalmers.se/research/group/logic/TypesSS05/`
   `Extra/geuvers.pdf`

3. Jean-Yves Girard; P. Taylor; Yves Lafon. Proofs and Types
   `http://www.paultaylor.eu/stable/prot.pdf`

Chapter 2

Hello, Haskell!

Basic expressions and functions

> 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 of this book, you will need to install the tools necessary to work with the language in order to complete the exercises as you work through the book. You can find the installation instructions online at `https://github.com/bitemyapp/learnhaskell`. If you wish, you can complete the installation and practice only for GHCi now and wait to install Cabal, but you will need Cabal up and running for the chapter about building projects. 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 writing and compiling 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 short for Read-Eval-Print Loop. REPLs are interactive programming environments where you can input code, have it evaluated

by the language implementation, and see the result. They originated with `Lisp` but are now common to modern programming languages including Haskell.

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

```
GHCi, version 7.8.3
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude>
```

It might not be exactly the same, depending on your configuration, but it should be very similar. If you are just downloading GHC for the first time, you may have the newest release number, which is 7.10.1. The newest version includes some important changes which we will cover in due time.

Now try entering some simple arithmetic at your prompt:

```
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`.

Abbreviating GHCi commands   Throughout the book, we'll be using GHCi commands, such as `:quit` and `:info` in the REPL. 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 as you remember the commands, we will start abbreviating them and you will, too.

Saving scripts and loading them

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, so if you don't understand what types or type signatures are, that's okay — we will explain them soon. For now, keep going.

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

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

```
Prelude>
```

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

Note that we used the command `:load` to load the file. Special commands that only GHCi understands begin with the `:` character. `:load` is *not* Haskell code; it's just a GHCi feature. We will see more of these commands throughout the book.

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 of smaller expressions. Declarations we'll cover more later, but it suffices to say for now that they are top-level bindings which lets us give names to expressions which we can use 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 where you can enter the code at the prompt, then hit '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 at the prompt, enter `ghci` to get your REPL going and 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.

2.4 Functions

Expressions are the most basic unit of a Haskell program, and functions are 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 (or parameter) 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 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, and it will be addressed in greater detail later.

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'd 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 wanted wherever we called that function.

For example, say you had a bunch of simple 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 the 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: what part is common to all the expressions? What 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 proper 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 arguments or parameters of the function, separated only by white space. Next there is an equal sign, which, notably, does not imply equality of value. Finally there is an expression that is the body of the function and can be evaluated to return a value.

Defining functions in a normal Haskell source code file and in GHCi are a little different. To introduce definitions of values or functions in GHCi you must use `let`, which looks like this:

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

In a source file we would enter it like this:

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

Let's examine each part of that:

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

1. Name of the function we are defining. Note that it is lowercase.

2. Argument to our function. The arguments to our function correspond to the "head" of a lambda.

3. The = is used to define (or *declare*) values and functions. Reminder: this is *not* how we express equality between two values in Haskell.

4. Body of the function, an expression that could be evaluated if the function is applied to a value. What `triple` is applied to will be the value which the argument x is bound to. 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! Names of modules and names of types, such as `Integer`, start with a capital letter. They can also be `CamelCase` when you want to maintain readability of multiple-word names.

Function names start with lowercase letters. Sometimes for clarity in function names, you may want `camelBack` style, and that is good style provided the very first letter remains lowercase.

Variables are lowercase.

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

```
Prelude> triple 2
6
```

Next, enter the second version (the one without `let`) 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.

Evaluating functions

Calling the function by name and introducing a value for the x argument makes our function a reducible expression. In a pure functional language like Haskell, we can replace applications of functions with their definitions and get the same result, just like in math. As a result when we see:

```
triple 2
```

We can know that, since triple is defined as `x = x * 3`, the expression is equivalent to:

```
 triple           2
(triple x = x * 3) 2
(triple 2 = 2 * 3)
2 * 3
6
```

What we've done here is apply `triple` to the value 2 and then reduce 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.

You may notice that after loading code from a source file, the GHCi prompt is no longer `Prelude>`. To return to the `Prelude>` prompt, use the command `:m`, which is short for `:module`. This will unload the file from GHCi, so the code in that file will no longer be in scope in your REPL.

Conventions for variables

Haskell uses a lot of variables, and some conventions have developed. It's not critical that you memorize this, because for the most part, these are merely conventions, but familiarizing yourself with them will help you read Haskell code.

Type variables (that is, variables in type signatures) generally start at a and go from there: a, b, c, and so forth. You may occasionally 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 have need to say it aloud. Usually this denotes a function that is closely related to or a helper function to 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, or a function that fetches some text might be called txt. Variables do not have to be a single letter, though they often are.

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 things you have named x, by convention that will usually be called xs, that is, the plural of x. You will see this convention often in the form `(x:xs)`, which means you have a list in which the head of the list is x and the rest of the list is xs.

All of these, though, are merely 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.

Intermission: Exercises

1. Given the following lines of code as they might appear in a source file, how would you change them to use them directly in the REPL?

```
half x = x / 2

square x = x * x
```

2. Write one function that can accept one argument and work 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)
```

2.5 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 just 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 which 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 mathematical 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 in an infix or prefix style, with a small change in syntax:

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

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 and tells you whether it's an infix operator with precedence. We will focus on the precedence and associativity for infix operators. Here's what the code in Prelude says for `(*)`, `(+)`, and `(-)` at time of writing:

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

:info (+) (-)
infixl 6  +, -
--         [4]
```

1. `infixl` means infix operator, 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

4. We use the comma here to assign left-associativity and precedence 6 for two functions `(+)` and `(-)`

Here `infixl` means the operator associates to the left. Let's play with parentheses and see what this means. Continue to follow along with the code via the REPL:

```
-- this
2 * 3 * 4

-- is evaluated as if it was
(2 * 3) * 4
-- Because of left-associativity from infixl
```

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

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

1. `infixr` means infix operator, 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.

It was hard to tell before why associativity mattered with multiplication, because multiplication is associative. So shifting the parentheses around never changes the result. Exponentiation, however, is not associative and thus makes a prime candidate for demonstrating left vs. right associativity.

```
Prelude> 2 ^ 3 ^ 4
2417851639229258349412352
Prelude> 2 ^ (3 ^ 4)
2417851639229258349412352
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?

Intermission: Exercises

Below are some pairs of functions that are alike except for parenthesization. Read them carefully and decide if the parentheses change the results of the function. 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)`

2.6 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 specified, no matter what order they appear in. 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> let y = 10
Prelude> let x = 10 * 5 + y
Prelude> let myResult = x * 5
```

As we've seen above when we worked with the `triple` function, we have to use `let` to declare something in the REPL.

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

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

To declare the same values in a file, such as `learn.hs`, we write the following:

```
-- learn.hs

module Learn where
-- 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.

x = 10 * 5 + y

myResult = x * 5

y = 10
```

Remember module names are capitalized, unlike function names. Also, in this function name, we've used camelBack case: the first letter is still lowercase, but we use an uppercase to delineate a word boundary for 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 indentation of Haskell code is significant and can change the meaning of the code. Incorrect indentation of code can also break your code. Reminder: 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 whitespaces at the end of lines of code, is considered bad style.

In source code files, indentation is important and often replaces syntactic markers like curly brackets, 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

-- Note that this code won't work directly in a
-- source file without embedding 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:

```haskell
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, try to pay careful attention to the indentation patterns as we have them printed. There are many cases where improper indentation will actually cause code not to work.

Also, when you write Haskell code, we reiterate here that you want to use spaces and *not* tabs for indentation. Using spaces will save you a nontrivial amount of grief. If you don't know how to make your editor only use spaces for indentation, *look it up*!

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

```haskell
module Learn where
-- First declare the name of our module so it
-- can be imported by name in a project.
-- We won't do this for a while yet.

x = 10
* 5 + y

myResult = x * 5

y = 10
```

You might see an error like:

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

code/learn.hs:10:1: parse error on input '*'
Failed, modules loaded: none.
```

Note that the first line of the error message tells you where the error occurred: **code/learn.hs:10:1** indicates that the mistake is in line 10, 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 that 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, and generally you should reserve 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 code/learn.hs
[1 of 1] Compiling Learn

code/learn.hs:11:1: parse error on input 'myResult'
Failed, modules loaded: none.
```

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 almost never 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
-- First declare the name of our module so it
-- can be imported by name in a project.
-- We won't do this for a while yet.

x = 10 * 5 + y

myResult = x * 5

y = 10
```

will cause this error:

```
Prelude> :r
[1 of 1] Compiling Main

code/learn.hs:7:1: parse error on input 'module'
Failed, modules loaded: none.
```

Note again that it says the parse error occurred at the beginning of the module declaration, but the issue is actually that `-` learn.hs had only one `-` when it needed 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, modules loaded: Learn.
Prelude> x
60
Prelude> y
10
Prelude> myResult
300
```

Intermission: Exercises

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

1. ```
   let area x = 3. 14 * (x * x)
   ```

2. ```
   let double x = b * 2
   ```

3. ```
   x = 7
    y = 10
   f = x + y
   ```

2.7 Arithmetic functions in Haskell

Let's work with a wider variety of arithmetic functions in Haskell. We will use some common operators and functions for arithmetic. The infix operators, their names, and their use in Haskell are listed here:

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

Always use `div` for integral division unless you know what you're doing. However, if the remainder of a division is the value you need, `rem` is usually the function you want. We will look at `/` in more detail later, as that will require some explanation of types and typeclasses.

2.8 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 just fine:

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

However, this will not work in some cases:

```
Prelude> 1000 + -9
<interactive>:3:1:
    Precedence parsing error
        cannot mix '+' [infixl 6] and
        prefix `-` [infixl 6]
            in the same infix expression
```

Fortunately, we were told about our mistake before any of our code was executed. Note how the error message tells you the problem has to do with precedence. Addition and subtraction have the same precedence (6), and GHCi 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 - 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 is so-called because while it can make the typing or reading of the code nicer, it changes nothing about the

semantics of our programs and doesn't change how we solve our 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 syntax sugar means the operator now has two possible interpretations. The two possible interpretations of the syntactic `-` are 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 a case of `-` being used for subtraction:

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

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

2.9 Parenthesizing infix functions

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. We will see more examples of the former case later in the book. For now, 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 `-` 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 that, 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> let x = 5
Prelude> let y = (1 -)
Prelude> y x
-4
```

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

2.10 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 about quot/rem and div/mod.[1]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 bit:

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

```
Given x is 10 and y is (-4)
```

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

---

[1]From Lennart Augustsson's blog `http://augustss.blogspot.com/`

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

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

```
10 == x == yeppers.
```

Now for div/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 == yeppers.
```

2.11   Evaluation

When we talk about reducing an expression, we're talking about evaluating the terms until it 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 which defers evaluation of terms until they're forced by other terms referring to them. We will return to this concept several times throughout the book as it takes time to fully understand.

Values are irreducible, but applications of functions to arguments are reducible. Reducing an expression means evaluating the terms until you're left with an irreducible value. As in the lambda calculus, application is evaluation, another theme that we will return to throughout the book.

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

```
1
"Icarus"
```

The following expressions can be reduced (evaluated, if you will) to a value:

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

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

2.12 Let and where

We can use `let` and `where` to introduce names for expressions.

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, modules loaded: FunctionWithWhere.
Prelude> printInc 1
3
Prelude>
```

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

```haskell
-- 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, modules loaded: FunctionWithLet.
Prelude> printInc2 3
5
```

If you loaded the `FunctionWithLet` version of `printInc2` in the same REPL session as `FunctionWithWhere` then it will have unloaded the old version before loading the new one. 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 Cabal and a `cabal repl` rather than GHCi itself.

Here's a demonstration of what is meant by GHCi unloading everything that had been defined in the REPL when you `:load` a file:

```
Prelude> :load FunctionWithWhere.hs
[1 of 1] Compiling FunctionWithWhere ...
Ok, modules loaded: FunctionWithWhere.
Prelude> print
print    printInc
Prelude> printInc 1
3
Prelude> :load FunctionWithLet.hs
[1 of 1] Compiling FunctionWithLet ...
```

```
Ok, modules loaded: FunctionWithLet.
Prelude> printInc2 10
12
Prelude> printInc 10

<interactive>:6:1:
    Not in scope: 'printInc'
    Perhaps you meant 'printInc2' (line 4)
```

`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.

Desugaring let to lambda

It turns out that `let` expressions have a fairly simple representation in lambdas. If we permit ourselves the writing of lambdas in Haskell using the provided lambda syntax, then we can transform the previous code in the following manner:

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

-- turns into

printInc2' n =
  (\plusTwo -> print plusTwo) (n + 2)
```

This doesn't work for every possible `let` expression as we don't have a good way to translate `let` expressions that use free variables recursively[2] into the lambda calculus. However, we want you to have a sense of how most things in Haskell can be translated into lambdas. It's all right if you

---

[2]Technically let and lambda are different primitives in GHC Haskell's underlying Core language. That's not the point, we're just trying to show you how programming languages can be lambdas under the hood.

don't understand right now why you'd care about this; we'll revisit this throughout the book.

Intermission: Exercises

Now for some exercises. First, determine in your head what the following expressions will return, then validate in the REPL:

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 into `where` clauses. You will have to give the value you're binding a name, although the name can be just a letter if you like. For example,

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

could be rewritten as

```
-- 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 the expressions are nested in that way, the equals signs do not have to line up. But notice we use a name that we will use to refer to this value in the REPL:

```
*Main> :l practice.hs
[1 of 1] Compiling Main
Ok, modules loaded: Main.
*Main> mult1
30
```

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

The lambdas beneath let expressions

It turns out that there's a fairly straightforward way to express lambdas in Haskell with the provided anonymous function syntax. Recall the identity function that we saw in the lambda calculus chapter:

```
λ x . x
```

Anonymous function syntax in Haskell uses a backslash to represent a lambda. It looks quite similar to the way it looks in the lambda calculus, and we can use it by wrapping it in parentheses and applying it to values:

```
Prelude> (\x -> x) 0
0
Prelude> (\x -> x) 1
1
Prelude> (\x -> x) "blah"
"blah"
```

We can also define the same function in the REPL using `let`:

```
Prelude> let id = \x -> x
Prelude> id 0
0
Prelude> id 1
1
```

Or we can define it this way:

```
Prelude> let id x = x
Prelude> id 0
0
Prelude> id 1
1
```

Let us translate a few **let** expressions into their lambda forms!

```
let a = b in c
```

```
-- equivalent to
```

```
(\a -> c) b
```

```
-- or a little less abstractly
```

```
let x = 10 in x + 9001
```

```
(\x -> x + 9001) 10
```

We can do a similar translation with **where**, although there are minute differences that we will not address here:

```
c where a = b
```

```
-- equivalent to
```

```
(\a -> c) b
```

```
-- Something more concrete again
```

```
x + 9001 where x = 10
```

```
(\x -> x + 9001) 10
```

We won't break down every single Haskell construct into the underlying lambdas, but try to keep this model in mind as you proceed.

More exercises!

Rewrite the following `let` expressions into declarations 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`

2.13 Chapter Exercises

The goal for all the following exercises is just 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

Here we've listed the information that GHCi gives us for various infix operators. We have left the type signatures in this time, although it is not directly relevant to the following exercises. 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 $
```

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 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.

Example of (\$) in some expressions:

```
Prelude> (2^) $ 2 + 2
16
Prelude> (2^) (2 + 2)
```

```
16
Prelude> (2^) 2 + 2
6
```

If you like, a way to understand `($)` in words is: "evaluate everything to the right of me first."

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 for ourselves why this code doesn't make sense if we examine the reduction steps.

```
-- Remember ($)'s definition
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)
-- This doesn't make sense, we can't apply 4
-- as if it was a function to the argument (*30)!
```

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
```

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

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 in your head 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 (remember: you'll need **let** when entering it directly into 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:

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

3. Now, what will happen if we enter this at our GHCi prompt. Try to reason out what 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 a function with a `where` clause in your source file. Load it into your REPL and make sure it still works as expected!

5. Now to the same source file where you have `waxOn`, add the `triple` function. Remember: You don't need `let` and 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 put in 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. Just spend some time getting comfortable with modifying the source file code, reloading it, and checking your modification in the REPL.

2.14 Definitions

1. An *argument* (also, parameter) is an input to a function. Where we have the function `f x = x + 2` which takes an argument and returns that value added to 2, x is the argument or parameter to our function.

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 this term.

3. A *redex* is a reducible expression.

4. 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.

5. A *function* is a mathematical object whose capabilities are limited to being applied to an argument and returning a result. Functions can be described as a list of ordered pairs of their 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.

6. *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`.

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

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

2.15   Follow-up resources

1. Haskell wiki article on Let vs. Where
   `https://wiki.haskell.org/Let_vs._Where`

2. Gabriel Gonzalez; How to desugar Haskell code
   `http://www.haskellforall.com/2014/10/how-to-desugar-haskell-code.html`

Chapter 3

Strings

Simple operations with text

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

3.1 Printing strings

So far we've been looking at doing arithmetic using simple expressions. In this section, we will look at another type of basic expression that processes 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 simple 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 and a bit of 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` 'character' and `String`. `String`s are lists 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 need to 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 in scope), so it has no way to know what the type of it 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. When we talk about lists in more detail later, we'll see why this is considered syntactic sugar; for now, we just need to understand that GHCi says "Hello!" has the type *list of Char*.

3.3 Printing simple strings

Now, let's look at some simple commands for printing strings of text in the REPL:

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

Here we've used the command `print` to tell GHCi to print the string to the display, so it does, with the quotation marks still around it.

Other commands we can use to tell GHCi to print strings of text into the display have slightly different results:

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

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

You can probably see that `putStr` and `putStrLn` are similar to each other, with one key difference. We also notice that both of these commands print the string to the display without the quotation marks. This is because, while they are superficially similar to `print`, they actually 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 run this code by loading it in GHCi and running `main`:

```
Prelude> :l print1.hs
```

```
[1 of 1] Compiling Print1
Ok, modules loaded: Print1.
Prelude> main
hello world!
Prelude>
```

The **main** function is the default function when you build an executable or run it in a REPL. As you can see, **main** has the type **IO ()**. **IO** — that's the letters I as in eye and O as in oh — stands for input/output but has a specialized meaning in Haskell. It is a special type used by Haskell when the result of running the program involves side-effects as opposed to being a pure function or expression. Printing to the screen is a side-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** function is the default executable function, this bit of code will be 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:

```
-- print2.hs
module Print2 where

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

And here's what you should see when you run this one:

```
Prelude> :l print2.hs
[1 of 1] Compiling Print2
Ok, modules loaded: Print2.
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.

You'll note the `putStrLn` function prints to the current line, then starts a new line, where `putStr` prints to the current line but doesn't start a new one. The `Ln` in `putStrLn` indicates that it starts a new line.

String concatenation

Concatenation, or to *concatenate* something, 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 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
-- The above line reads as: "myGreeting has the type String"
myGreeting = "hello" ++ " world!"
-- Could also be: "hello" ++ " " ++ "world!"
-- to obtain the same result.

hello :: String
hello = "hello"

world :: String
world = "world!"

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

If you execute this, you should see something like:

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

Remember you can use `:m` to return to Prelude if desired.

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 globally so that their

values are available to all functions in the module.

2. We specify explicit types for top-level definitions.

3. We concatenate strings with `(++)` and `concat`.

Global versus local definitions

What does it mean for something to be at the top level of a module? It means it is defined globally, not locally. To be locally defined would mean the declaration is nested within some other expression and is not visible to code importing the module. We practiced this in the previous chapter with `let` and `where`. Here's an example for review:

```haskell
module GlobalLocal where

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

topLevelValue :: Integer
topLevelValue = 5
```

In the above, you could import and use `topLevelFunction` or `topLevelValue` from another 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 more pleasant and less repetitious. Also note we explicitly declared the type of `woot` in the `where` clause. This wasn't necessary (Haskell's type inference would've figured it out fine), but it was done here to show you how in case you need to. Be sure to load and run this code in your REPL:

```
Prelude> :l Global.hs
```

```
[1 of 1] Compiling GlobalLocal
Ok, modules loaded: GlobalLocal.
*GlobalLocal> 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).

Intermission: Exercises

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

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

2. These lines of code are from a REPL session. Is h in scope for function g?

   ```
   Prelude> let f = 3
   Prelude> let 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
   ```

3.4 Type signatures 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]
```

(n.b., Assuming you are using GHC 7.10, if you check this type signature in your REPL, you will find an unexpected and possibly confusing result. We will explain the reason for it later in the book. For your purposes at this point, please understand `Foldable t => t [a]` as being `[[a]]`. The `Foldable t`, for our current purposes, can be thought of as another list. In truth, list is only one of the possible types here — types that have instances of the Foldable typeclass — but right now, lists are the only one we care about.)

But what do these types mean? Here's how we can break it 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]`, which is a list of elements whose type we don't yet know.

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 type of list, 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 `[Char]`. The type variable *a* in `[a]` is polymorphic. Polymorphism is an important feature of Haskell. For concatenation, every list must be the same type of list; 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* (a == a).

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

-- but

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

<interactive>:14:13:
    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 typeclass `Num` for the type Char. Unsurprisingly, Char isn't an instance of `Num`.

Intermission: Exercises

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"]`

3.5 An example of concatenation and scoping

We will use parentheses to call `++` as a typical 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 `myGreeting`, we moved `++` to the front like a typical prefix function invocation. Using it as a prefix function in the `secondGreeting`, though, forces us to shift some things around. Parenthesizing it that way emphasizes the right associativity of the `++` function.

The `where` clause creates local bindings for expressions that are not visible at the top level. In other words, the `where` clause in the main function introduces a definition visible only within the expression or function it's attached to, rather than making it globally 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, interpreted )

print3broken.hs:6:12: Not in scope: 'greeting'
Failed, modules loaded: none.
```

Let's take a closer look at this error:

```
print3broken.hs:6:12: Not in scope: 'greeting'
#              [1][2]     [3]            [4]
```

1. The line the error occurred on: in this case, line 6.

2. The column the error occurred 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 referred 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.

3.6 More list functions

You can use most functions for manipulating lists on strings as well in Haskell, because a string is just a list of characters, `[Char]`, under the hood.

Here are some examples:

```
Prelude> :t 'c'
'c' :: Char
Prelude> :t "c"
"c" :: [Char] -- List of Char is String, same thing.

-- the : operator, called "cons," builds a list
Prelude> 'c' : "hris"
"chris"
Prelude> 'P' : ""
"P"

-- head returns the head or first element of a list
Prelude> head "Papuchon"
'P'

-- 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"

-- drop returns the remainder of the list after the
-- specified number of elements has 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"

-- !! returns the element that is in the specified
-- position in the list. Note that this is an
-- indexing function, and indices in Haskell 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'
```

3.7 Chapter Exercises

Reading syntax

1. For the following lines of code, read the syntax carefully and decide if they are written correctly. Test them in your REPL after you've decided 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 came 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]]

Can you match each of the previous expressions to one of these results 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 following 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. Now 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 the argument to 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 `GlobalLocal` module from this chapter.

3. Write a function of type `String -> Char` which 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 the type signature and fill in 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 actually get you the fourth. Keep this in mind as you write this function.

4. Now change that function so the string input 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 a function called 'reverse' already in Prelude, so if you call 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 quite possible using only what we've learned so far. First write it as a single function in a source file. This doesn't need, 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 each other. 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:

```haskell
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." That `rvrs` function plus its argument are now the argument to `print`. It's important to put them inside the parentheses so that that function gets applied and evaluted first, and then that result is printed.

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

3.8   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 it or *inhabit* it. 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 is `[Char]`. The *concatenation* function in Haskell is `(++)` which has type `[a] -> [a] -> [a]`. For example:

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

4. *Scope* is where a variable referred to by name is valid. Another word used with the same meaning are *visibility*, because if a variable isn't *visible* it's not in *scope*.

5. *Local bindings* are bindings local to particular expression. The primary delineation here from *global* bindings is that *local* bindings cannot be imported by other programs or modules.

6. *Global* or top level bindings in Haskell mean bindings visible to all code within a module and, if made available, can be imported by other modules or programs. Global bindings in the sense that a variable is unconditionally visible throughout an entire program do not exist in Haskell.

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

Chapter 4

Basic datatypes

Because pigs can't fly

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

Types are a powerful means of classifying, organizing, and delimiting data to only the forms we want to process in our programs. Types 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 built-in datatypes that categorize and delimit values. While those are useful datatypes and cover a lot of types of values, they certainly 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 typeclasses

4.2 Anatomy of a data declaration

Data declarations are how datatypes are defined.

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 simple datatype to see how datatypes are structured and get acquainted with the vocabulary. `Bool` isn't a datatype we've seen

yet in the book, but it provides for truth values. Because there are only two truth values, it's easy to list all the data constructors:

```
-- the definition of Bool
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. Pipe | indicates logical disjunction, "or." So, a Bool value is True *or* False.

4. Data constructor for the value True.

The 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 name of the type that will appear in type signatures), the equals sign to denote a definition, and then data constructors (or names of values that inhabit your term-level code).

Let's look at a very simple illustration of where the different parts 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, in our code:

```
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.

Intermission: Exercises

Given the following datatype, answer the following questions:

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

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 there?

3. We are trying to write a function `changeMood` to change Chris's mood instantaneously. 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
   ```

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.

4.3 Numeric types

Let's next look at numeric types, because we've already seen these types just a bit in previous chapters, and numbers are familiar territory. It's

important to understand that Haskell does not just use one type of number. For most purposes, the types of numbers we need to be concerned with are:

Integral numbers   : 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.

Fractional   : These are not integers. Fractional numbers include the following four types:

1. `Float`: This is the type used for single-precision floating point numbers. Where fixed point number representations have immutable numbers of digits assigned for the parts of the number before and after the decimal point, 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` will be a value carrying 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. `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 quite unlikely.

These numeric datatypes all have instances of a typeclass called `Num`. We will talk about typeclasses in the upcoming chapters, but as we look at the types in this section, you will see Num listed in some of the type information. Typeclasses are a way of adding functionality to types that is reusable across all the types that have instances of that typeclass. Num is a typeclass 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 typeclass 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.

Integral numbers

As we noted above, there are two types of integral numbers: Int and Integer.

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 straightforward enough; for the most part, they are the sorts of numbers we're all used to working with in arithmetic equations that involve whole numbers. 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 towards negative infinity, zero, and recursive constructors headed towards positive infinity. This representation would be terribly inefficient so there's some GHC magic sprinkled on it.

Why do we have Int?

The Int numeric type is an artifact of what computer hardware has supported natively over the years. Most programs should use Integer and 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

<interactive>:11:1: Warning:
    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 first computation is fine, because it is within the range of valid values of type Int8, but the `127 + 1` overflows 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. The representation used for the fixed-size Int types is *two's complement.* Here the 8 represents how many bits the type uses to represent integral numbers. 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 the `maxBound` and `minBound` functions. Here's an example using our Int8 and Int16 example:

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

These functions are from the typeclass `Bounded`, and they will work to tell

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

Fractional numbers

The four common Fractional types in use in Haskell are `Float`, `Double`, `Rational`, and `Scientific`. Rational, Double, and Float come with your install of GHC. Scientific comes from a library. Rational and Scientific are arbitrary precision, with the latter being much 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 typeclass constraint. You can read it as "the type variable a must implement the Fractional typeclass." This type information is telling us that whatever type of number a turns out to be, it must be a type that has an instance of the Fractional typeclass. 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 typeclass that requires types to already have an instance of the Num typeclass. We describe this relationship between typeclasses by saying that Num is a superclass of Fractional. So `(+)` and other functions from the Num typeclass can be used with Fractional numbers, but functions from the Fractional typeclass cannot be used with all instances of Num:

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

```
Prelude> 1 / 2
0.5
Prelude> 4 / 2
2.0
Prelude>
```

Note that even when we had a whole number as a result, the number was printed as `2.0`, despite having no fractional component. 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're usually better off using the arbitrary precision sibling to Integer, Scientific, which is available in a library on Hackage.[1] Most people do not find it easy to reason about floating point arithmetic and find it difficult to code around the quirks (those quirks are actually by design, but that's another story), so in order to avoid making mistakes just use arbitrary-precision types as a matter of course.

4.4   Comparing values

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

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

---

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

Notice here that we first declared 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. Bool gets its name from boolean logic. 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 typeclass constraints again. `Eq` is a typeclass that includes everything that can be compared and determined to be equal in value; `Ord` is a typeclass 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 typeclass constraint allows for the maximum amount of flexibility. These equality and comparison functions can take any values that can be said to have equal 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 Bool value. 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
```

These examples are easy enough to understand. 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 was greater because B == B, then "Brack" would've been greater because r > a in the lexicographic ordering Haskell uses for characters. Note that this is leaning on the Ord typeclass instances for list *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 = Blah | Woot deriving Show
Prelude> [Blah, Woot]
[Blah,Woot]
Prelude> [Blah, Woot] > [Woot, Blah]

<interactive>:28:14:
    No instance for (Ord Mood) arising from a use of '>'
    In the expression: [Blah, Woot] > [Woot, Blah]
    In an equation for 'it':
      it = [Blah, Woot] > [Woot, Blah]
```

Here is another thing that doesn't work with these functions:

```
Prelude> "Julie" == 8

<interactive>:38:12:
    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 admitted values of matching types. Once you've given a term-level value that is a string such as "Julie," the type is determined and the other argument must have the same type. The error message we see above is telling us that the type of the literal value '8' doesn't match the type of the first value, and for this function, it must.

Go on and Bool me

In Haskell the Bool datatype comes standard in the Prelude. Bool is a sum type with two constructors. For now, it's enough to understand that sum type means *or*, or logical disjunction. We will look more closely at sum types in a later chapter.

In Haskell there are six categories of entities that have names. First, there are variables and data constructors which exist at the term-level. 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 & verification of your program, we have type variables, type constructors, and typeclasses. Lastly, for the purpose of organizing our code into coherent groupings across different files (more later), we have modules which have names as well.

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

This declaration creates a datatype with the type constructor `Bool`, and we refer to specific types by their type constructors. We 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, as you cannot specify in the type that it only accept one specific value. An English language formulation of this datatype would be something like:

```
data Bool = False | True
-- The datatype Bool is represented by the values True or False.
```

Now let's have some fun with Bool in action. 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> 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 we do use Bool and these associated functions?

```
-- && is conjunction, so this means "true and true."
Prelude> True && True
True
-- || is disjunction, so this means "false or true."
Prelude> False || True
True
Prelude> True && False
False
Prelude> not True
False
Prelude> not (True && True)
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. What is built into Haskell's Prelude module is significant because everything in it is automatically imported and in scope. We will demonstrate how to shut this off later, but for now, this is what you want.

Intermission: Exercises

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!"`

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> if True then "Truthin'" else "Falsin'"
"Truthin'"
Prelude> if False then "Truthin'" else "Falsin'"
"Falsin'"
Prelude> :t if True then "Truthin'" else "Falsin'"
if True then "Truthin'" else "Falsin'" :: [Char]
```

The structure here is `if CONDITION then EXPRESSION_A else EXPRESSION_B`. If the condition (which must evaluate to `Bool`) reduces to the Bool value `True`, then `EXPRESSION_A` is the result, otherwise

EXPRESSION_B. Here the type was String (aka [Char]) because that's the type of the value that is returned as a result.

If-expressions can be thought of as a way to choose between two values. You can embed a variety of expressions within the `if` 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> let x = 0
Prelude> if (x + 1 == 1) then "AWESOME" else "wut"
"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> let x = 0
Prelude> if (x * 100) then "adopt a dog" else "or a cat"
```

```
<interactive>:15:7:
    No instance for (Num Bool) arising
    from a use of '*'

    In the expression: (x * 100)
    In the expression:
       if (x * 100)
         then "adopt a dog"
         else "or a cat"
    In an equation for 'it':
        it = if (x * 100)
               then "adopt a dog"
               else "or a cat"
```

We got 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
typeclass. To oversimplify, `(x * 100)` evaluates to a numeric result, and
numbers aren't truth values. It would have typechecked had the condition
been `x * 100 == 0` or `x * 100 == 9001`. In those cases, it would've been
an equality check of two numbers which 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"
```

If you test this in the REPL, it should play out like this:

```
Prelude> :l greetIfCool1.hs
[1 of 1] Compiling GreetIfCool1
Ok, modules loaded: GreetIfCool1.
Prelude> greetIfCool "downright frosty yo"
eyyyyy. What's shakin'?
Prelude> greetIfCool "please love me"
pshhhh.
```

Also note that **greetIfCool** could've been written with **cool** being a function rather than 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"
```

4.5   Tuples

The tuple in Haskell is a type that allows you to store and pass around multiple values within a single 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 how many constituents are in each tuple: the two-tuple or pair, for example, has two values inside it `(x, y)`; the three-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 two-tuple, a tuple with two constituents. The two-tuple is expressed at both the type level and term level with the constructor `(,)`.

The two-tuple in Haskell has some default convenience functions for getting the first or second value. They're named `fst` and `snd`:

```
fst :: (a, b) -> a
snd :: (a, b) -> b
```

As you can see from the above types, there's nothing those functions could do other than return the first or second value respectively. Also note that there are two polymorphic type variables, a and b which are allowed to be different types. So, a two-tuple can have type (Integer, String) or (Integer, Integer) or anything else you can imagine for the types of a and b.

Here are some examples of manipulating tuples, specifically the two-tuple:

```
Prelude> let 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 had 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.

It's generally unwise to use tuples of an overly large size, both for effiency and sanity reasons. Most tuples you see will be `( , , , , )` (5-tuple) or smaller.

4.6 Lists

Lists in Haskell are another type used to contain multiple values within a single value. However, they differ from tuples in three important ways: First, all constituents 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 constituents within a list can change as you operate on the list, unlike tuples where the arity is set in the type and immutable.

Here's an example for your REPL:

```
Prelude> let awesome = ["Papuchon", "curry", "Haskell"]
Prelude> awesome
["Papuchon","curry","Haskell"]

Prelude> :t awesome
awesome :: [[Char]]
```

First thing to note is that `awesome` is a list of a list of Char values because it is a list of strings, and a string is itself just a type *alias* for [Char]. A type alias, or type synonym, is what it sounds like: we use one name for a type, usually for some convenience, that has another name. Here "string" is an alias or synonym for a *list of characters*. 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> let alsoAwesome = ["Quake", "The Simons"]
Prelude> :t (++)
(++) :: [a] -> [a] -> [a]
Prelude> awesome ++ alsoAwesome
["Papuchon","curry","Haskell","Quake","The Simons"]

Prelude> let allAwesome = [awesome, alsoAwesome]
Prelude> allAwesome
[["Papuchon","curry","Haskell"], ["Quake","The Simons"]]
Prelude> :t allAwesome
allAwesome :: [[[Char]]]
Prelude> :t concat
concat :: [[a]] -> [a]
Prelude> concat allAwesome
["Papuchon","curry","Haskell","Quake","The Simons"]
```

We're going to go into much more detail about lists and operations with them in an upcoming chapter, because there is a lot you can do with this datatype.

4.7 Chapter Exercises

As in previous chapters, you will gain more by working out the answer before you check what GHCi tells you, but be sure to use your REPL to check your answers to the following exercises. Also, you will need to have the awesome, alsoAwesome, and allAwesome code from above in scope for this REPL session. For convenience of reference, here are those values again:

```
awesome = ["Papuchon", "curry", "Haskell"]
alsoAwesome = ["Quake", "The Simons"]
allAwesome = [awesome, alsoAwesome]
```

length is a function that takes a list and returns a result that tells how many items are in the list.

1. Given the definition of `length` above, what would the type signature be? How many arguments, of what type does it take? 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 these two expressions. One works and one returns an error. Determine which will return an error and why.

    (n.b., If you're checking the type signature of `length` in GHC 7.10, you will find `Foldable t => t a` representing `[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. We will explain it in detail later.)

    ```
    Prelude> 6 / 3
    -- and
    Prelude> 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:

    ```
    Prelude> let x = 5
    Prelude> x + 3 == 5
    ```

7. Below are some bits of code. Which will work? Why or why not? If they will work, what value would these reduce to?

```
Prelude> length allAwesome == 2
Prelude> length [1, 'a', 3, 'b']
Prelude> length allAwesome + length awesome
Prelude> (8 == 8) && ('b' < 'a')
Prelude> (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 just 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 if-then-else

```
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
```

Reading 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 $[1, 2, 3]$. Hint: you may need to refer back to the section about variables conventions in "Hello Haskell" to refresh your memory of this notation.

```
\ x : xs -> x
```

4. 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`

4.8   Definitions

1. A *tuple* is an ordered grouping of values. In Haskell, there is no singleton tuple, but there is a zero tuple also called *unit* or `()`. The types of the elements of tuples are allowed to vary, so you can have both (String, String) or (Integer, String). Tuples in Haskell are the usual means of expressing an anonymous product.

2. A *typeclass* is a set of operations defined with respect to a polymorphic type. When a type is an instance of a typeclass, values of that type can be used in the standard operations defined for that typeclass. In Haskell typeclasses are unique pairings of class and 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 just 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 to create values that inhabit that type, data declarations 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
-- creates a new type alias Name of the
-- type String *not* a data declaration,
-- just a type alias declaration
```

7. *Arity* is the number of arguments a function accepts. This notion is a little slippery in Haskell as, due to currying, all functions are 1-arity and we handle accepting multiple arguments by nesting functions.

8. *Polymorphism* in Haskell means being able to write code in terms of values which may be one of several, or any, type. Polymorphism in Haskell 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`:

```
isEqual :: Eq a => a -> a -> Bool
isEqual x y = x == y
```

Is polymorphic, but *constrained* or *bounded* to the set of types which have an instance of the Eq typeclass. The different kinds of polymorphism will be discussed in greater detail in a later chapter.

4.9 Answers

Intermission Exercises

1. Given the following datatype, answer the following questions:

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

   a) The type constructor is Mood.

   b) Possible term-level values of type Mood are Blah and Woot.

   c) The type signature `changeMood :: Mood -> Woot` is wrong because at the type level we can't specify values (or data constructors), only the types. So our type would have to go Mood -> Mood.

   d) The type constructor, Mood, shouldn't be used as a value, so that had to be fixed first. Since the goal was to change the mood, it should look this:

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

   You might have changed the underscore to Woot, and for this exercise that's fine. The underscore is an "otherwise" case that will catch any other inputs that aren't Blah, which doesn't matter for this exercise, but might if you were going to someday enlarge the number of moods you have encoded in your datatype.

   e) Your answer for this one comes from GHCi.

2. The following lines of code may have mistakes — some of them won't compile! There isn't necessarily only one right answer to each of these, but if the answer here was different from the way you did it, and your answer still compiles and runs, then you did fine.

   a) `not True && True`

   b) Assuming you have an x in scope, `not (x == 6)`.

   c) `(1 * 2) > 5` — fine as is

   d) `"Merry" > "Happy"`

e) `"1, 2, 3" ++ "look at me!"` — the point here is that they weren't the same type of values in the list, so they couldn't be concatenated. Any way you changed it to make them the same type is fine.

Chapter Exercises

1. For this first set, you needed these things in scope:

```
awesome = ["Papuchon", "curry", "Haskell"]
alsoAwesome = ["Quake", "The Simons"]
allAwesome = [awesome, alsoAwesome]
```

a) `length` takes one list argument (or Foldable argument) and returns an Int value. It is defined with an Int result because Int is definitely finite while Integer is not.

b) What are the results of the following expressions? You should have decided what the result would be and then verified your hypothesis in the REPL, so answers shouldn't be necessary here.

c) One works and one returns an error. The second one returns the error because the result of `length` is an Int, not a Fractional number.

```
Prelude> 6 / 3
-- and
Prelude> 6 / length [1, 2, 3]
```

d) How can you fix the broken code from the preceding exercise using a different division function/operator? Again, you should have been able to verify this in the REPL, but if you're stuck, here's one possible answer:

```
div 6 (length [1, 2, 3])
```

e) What is the type of the expression `2 + 3 == 5`? What would we expect as a result? This one you should have thought about, tried to answer in your head and verified in the REPL:

```
Prelude> :t (2 + 2 == 5)
```

f) Again, use your REPL to verify your ideas here by querying the type and checking the results.

g) Below are some bits of code. Which will work? Why or why not? Again, the goal here is to think through them then verify in the REPL. GHCi is unlikely to give you an incorrect answer, assuming you input everything correctly.

h) 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 just what it sounds like.

```haskell
isPalindrome :: (Eq a) => [a] -> Bool
isPalindrome x = x == reverse x
```

i) Write a function to return the absolute value of a number using if-then-else:

```haskell
myAbs :: Integer -> Integer
myAbs x = if (x < 0) then (negate x) else x
```

j) Fill in the definition of the following function, using `fst` and `snd`:

```haskell
f :: (a, b) -> (c, d) -> ((b, d), (a, c))
f x y = ((snd x, snd y), (fst x, fst y))
```

2. Reading syntax

a) A function that adds 1 to the length of a string argument and returns that result. One possible way to do it:

```haskell
x = (+ 1)

f xs = x w
    where w = length xs
```

b) The identity function, `id`:

```haskell
id x = x
```

c) When fixed, this function will return 1 from the value [1, 2, 3]. Hint: you may need to refer back to the section about variables conventions in "Hello Haskell" to refresh your memory of this notation.

```haskell
f (x:xs) = x
```

d) When fixed, this function will return 1 from the value (1, 2)

```
f (a, b) = a
```

3. Match the function names to their types All of these can be easily
   verified in the REPL.

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

Haskell has a robust and expressive type system. Types play an important role in the readability, safety, and maintainability of Haskell code. In the last chapter, we looked at some built-in datatypes, such as `Bool` and tuples, and we also looked just a bit at the typeclasses `Num` and `Eq`. In this section, we're going to take a deeper look at the type system and

- learn more about querying and reading type signatures;

- see that currying has, unfortunately, 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?

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 "something in common" 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[1] sense.

As we saw in the last chapter, data constructors are the values of a particular type; they are also functions that let us create data, or values, of a particular type, as will become more clear in a later chapter. In Haskell, you cannot create untyped data, so except for a sprinkling of syntactic sugar for things like numbers or functions, everything originates in a data constructor from some definition of a type.

---

[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.
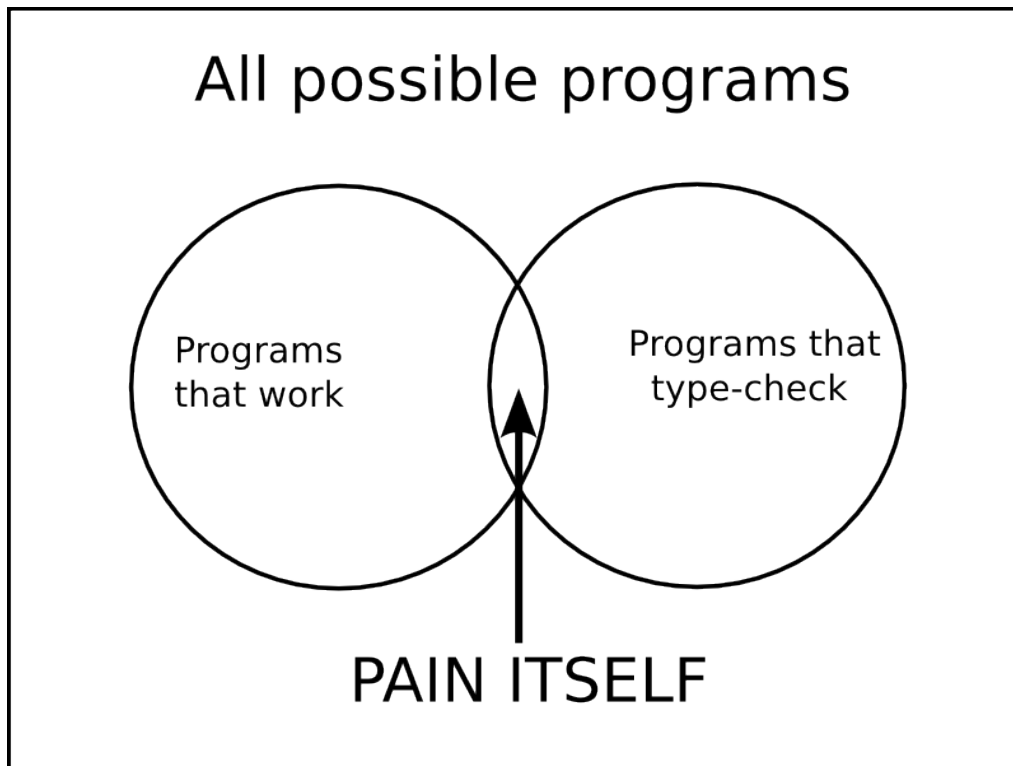
Why pain? Because the overlap is so small!

This type system makes life difficult for us from *two* different directions. Not only does it allow bad (that is, not well-typed) programs, but the types don't let us express as many programs that *should* work! Many, perhaps most, programming languages have type systems that feel like arguing or bartering 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 with your type system.

An improved type system, in contrast to the one above, looks more like this:
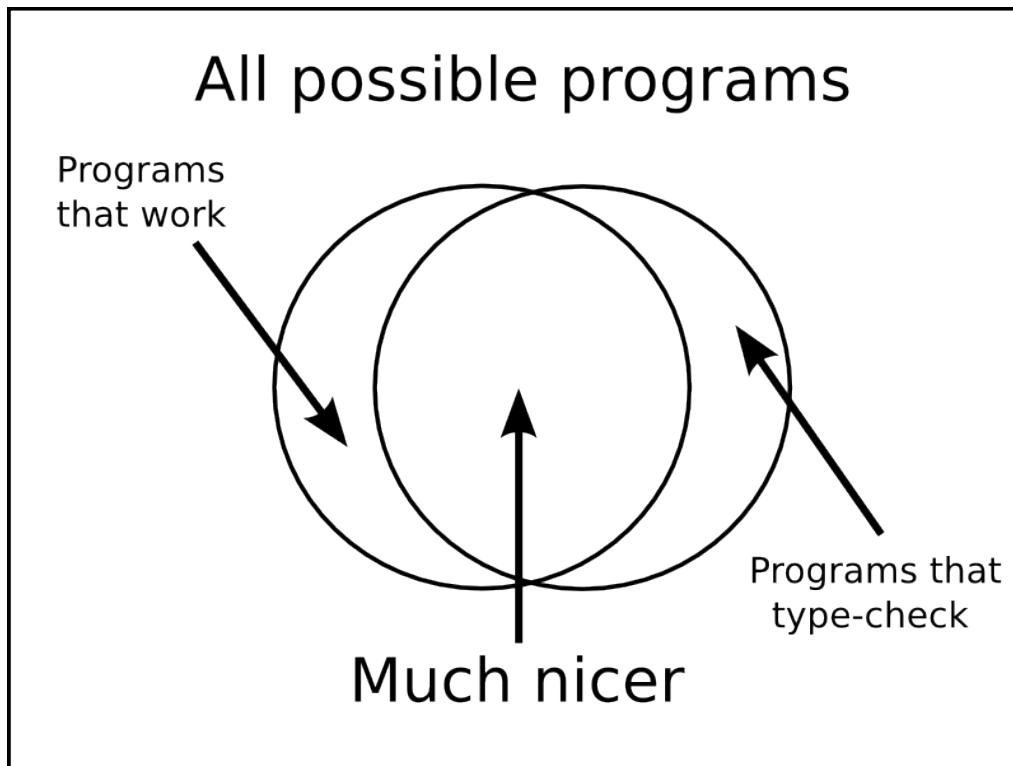
The above is an example of what we would actually like, a type system which, as much as is possible, lets us only express programs that work and express *many* possible working programs.

Strong typing means that we are guaranteed not to have certain sorts of errors in our code. For example, if a function expects an integral value, you cannot put a string or some other type of value in that spot; it will not typecheck and it will not compile.

Static typing means that the types are known to the compiler and checked for mismatches, or type errors, at compile time. Code that typechecks at compile time may still break at runtime, but checking for type errors at compile time significantly reduces the risk of programs breaking at runtime. You may at first feel that the strong, static type system requires a lot of upfront work. As we will see, however, 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, which can help tremendously.

Haskell's type system allows for a nifty feature known as type inference. We can declare our types when we write our programs, but the compiler will infer the types for expressions that have no declared type. It is better to have explicit type declarations in any nontrivial program, but type inference can be helpful as you're learning and experimenting with writing new programs.

An example of a type is **Bool**, which we remember from the last chapter. The **Bool** type is a set with two inhabitants, **True** and **False**. Any time the value **True** or **False** occurs in a Haskell program, the typechecker will know they're members of the **Bool** set or type. The inverse is that anytime type **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.

5.3   Querying and Reading Types

In previous chapters, we've seen that we can query types in the REPL by use of the **:type** or **:t** command. 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 typeclass information because the compiler doesn't know which specific numeric type a value is until the type is either declared (more on that later) 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 `Num a => a` value:

```
Prelude> :type 13
13 :: Num a => a
```

Now, let's look at the type signatures from some of the functions we've already learned:

```
Prelude> :type not
not :: Bool -> Bool
```

The `->` you keep seeing in type signatures 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
```

Since it has no data constructors, the value of type `->` that shows up at term-level is the function. *Functions are values.*

If you compare this to the first line of `:info` of the type constructor for the two-tuple, you see the similarity:

```
Prelude> :info (,)
data (,) a b = (,) a b
```

The function `fst` is a value of type `(a, b) -> a` where `->` is an infix type that takes two arguments:

```
fst :: (a, b)  ->    a
--       [1]   [2]  [3]
```

1. Argument to `fst`, has the type `(a, b)`. Note that the tuple type itself `(,)` takes two arguments `a` and `b` here.

2. The function type, `->`. Here it's taking two arguments. One is the argument `(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)`.

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 types of values are inside the list.

Type signatures in GHC 7.10 The last release of GHC brought noticeable changes, including changes to type signatures. We aren't going to explain it in detail at this point (a later chapter on Foldable and Traversable will cover the change in depth). In GHC 7.10, the type of `length` is now `Foldable a => t a -> Int` where `Foldable` is a typeclass. This means `length` can now be used on any type that has an instance of the `Foldable` class, not only on lists. It does not affect how `length` will behave with lists, and you can easily force `length` to specialize from a constrained-polymorphic `t a` to a `[a]`. Until we've covered typeclasses in enough detail that this makes sense to you, we will continue to show the old type signatures. Note that this does not only affect `length` but many standard functions that we use with lists and strings.

5.4 Typeclass-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, and we'll explain why very soon.

As we saw with numeric datatypes, the compiler gives the least specific and most general type it can. Instead of limiting this function to a concrete type, we get a typeclass-constrained polymorphic type variable. We talked about typeclasses just a bit in the previous chapter, and we'll save a fuller explanation of them for the next chapter. What we need to know here is that `Num` and `Fractional` are typeclasses, not concrete types. Each typeclass offers a standard set of functions that can be used across several concrete types. When a typeclass is constraining a type variable in this way, the variable could represent any of the concrete types that have instances of that typeclass.

For now, we can think of the type `Num a => a` as saying, "variable `a` can have any type provided it satisfies the properties of being a number". This generalization of number-hood is what lets us use the same numerical literals to represent numeric values of different types. In fact, we can start with a `Num a => a` value and then create specific versions of it with a concrete 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 a `Num` instance:

```
Prelude> let fifteen = 15
Prelude> :t fifteen
fifteen :: Num a => a
Prelude> let fifteenInt = fifteen :: Int
Prelude> let fifteenDouble = fifteen :: Double
Prelude> :t fifteenInt
fifteenInt :: Int
Prelude> :t fifteenDouble
fifteenDouble :: Double
```

So we've shown that you can make a type more specific, we went from `Num a => a` to `Int` and `Double`. This works because `Int` and `Double` have instances of the `Num` typeclass:

```
Prelude> :info Num
[...irrelevant bits elided...]
instance Num Int -- Defined in 'GHC.Num'
instance Num Double -- Defined in 'GHC.Float'


Prelude> fifteenInt + fifteenInt
30
Prelude> fifteenDouble + fifteenDouble
30.0

-- BUT:

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
```

We can also make more specific versions of our `Num a => a` value named **fifteen** by simply using it in a way that requires it to become something more specific:

```
Prelude> fifteenDouble + fifteen
30.0
Prelude> fifteenInt + fifteen
30
```

A type signature might have multiple typeclass 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 -> Ordering
```

Here, the constraints look like a tuple but 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. They do represent a product, or conjunction, of constraints.

In the first example above, there are two constraints, one for each variable. Both `a` and `b` must have instances of the `Num` typeclass. 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`.

In either case, this isn't saying `a` must be a tuple or that the function will have a tuple at the term level, but that this intersection of constraints exists for the type.

Intermission: Exercises

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`

5.5 Currying

As stated earlier, arguments (*plural*) is 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.

The arrows we've been seeing in type signatures denote the function type. Each arrow represents one argument and one result, with the final type being the final result. We can elaborate on that in a few ways:

```
(+) :: Num a => a -> a -> a
--      |  1  |

(+) :: Num a => a -> a -> a
--             |  2  |

(+) :: Num a => a -> a -> a
--                        [3]
```

1. Typeclass constraint saying that `a` must have an instance of `Num`.

2. The boundaries of `2` demarcate what you might call the two "arguments" to the function `(+)`, but really, 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 nesting is more apparent when one realizes that `(->)` is the type constructor for functions and that `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 actually returning *another* function that accepts the next argument. This is called currying.

3. The result type for this function.

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. Because it associates to the right, types get parenthesized like so:

```
a -> a -> a

-- gets paren'd into

a -> (a -> a)

-- and

(a -> b) -> [a] -> [b]

-- gets paren'd into

(a -> b) -> ([a] -> [b])
```

Let's see if we can unpack the notion of a right-associating infix operator giving us curried functions. As we saw with the lambda calculus, *application is evaluation.* Remember, when we have a lambda expression that appears to have two arguments, 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 like when we apply a lambda expression to an argument, and then return a result that is a function and needs to be applied to a second argument.

Let's kick around currying a bit to see what it does for us:

```
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 is taking one argument and returning a function that takes one argument and returns one result:

```
Prelude> :t addStuff
addStuff :: Integer -> Integer -> Integer
Prelude> let addTen = addStuff 5
Prelude> let 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 described as *partial application*. This lets us reuse `addStuff` and create a new function from it with one of the arguments applied.

When one considers that `(->)` is a type constructor and associates to the right, this becomes more clear.

```
addStuff :: Integer -> Integer -> Integer

-- but with explicit parenthesization

addStuff :: Integer -> (Integer -> Integer)
```

The way you can read the explicitly parenthesized type for `addStuff` is, "I take an `Integer` argument and return a function that takes an `Integer` and returns an `Integer`". You saw this in action when we partially applied the `addStuff` function above.

Binding variables to types

Let's next look at an example of the effect that binding arguments has on types. We will declare a function with a number of arguments that share the same type, but don't get used:

```haskell
funcIgnoresArgs :: a -> a -> a -> String
funcIgnoresArgs x y z = "Blah"
```

Then we load this and apply the first argument in a few different ways to see what happens:

```
Prelude> :t funcIgnoresArgs
funcIgnoresArgs :: a -> a -> a -> String
-- Cool, this is the type we saw in the file


Prelude> :t funcIgnoresArgs (1 :: Integer)
funcIgnoresArgs (1 :: Integer) :: Integer
                                 -> Integer
                                 -> String
```

We lost one (`->`) because we applied one of our functions. The function `a -> a -> a -> String` is only conceptually, but not *actually* one function. Technically it's 3 functions nested one inside another. We could read it as `a -> (a -> (a -> String))`. One way to know how many function objects there are is by their type constructors, and (`->`) is the type constructor for functions. Let's run this same function through some drills to see what is meant.

```
-- `undefined' can pretend to be any type
-- put differently, inhabits all types
Prelude> :t undefined
undefined :: t
Prelude> let u = undefined

Prelude> :t funcIgnoresArgs u
funcIgnoresArgs undefined :: a -> a -> String
Prelude> :t funcIgnoresArgs u u
funcIgnoresArgs u u :: a -> String
Prelude> :t funcIgnoresArgs u u u
funcIgnoresArgs u u u :: String
Prelude> funcIgnoresArgs u u u
"Blah"
```

Here undefined, sometimes also called "bottom", provided no additional type information so the `a` in the type signature was never made more specific. The progression of type signatures went like so:

```
-- Remember
a -> a -> a -> String

-- is actually
a -> (a -> (a -> String))

-- So applying each argument
-- produces the following progression
a -> a -> a -> String

    a -> a -> String

        a -> String

            String
```

Consider this next example for a function of a different type but with the same behavior:

```
-- Spaced out for readability
diffFunc :: a  ->   b  ->   c   ->   String
--             [1]     [2]       [3]

-- is actually

diffFunc :: a  ->   (b  ->   (c   ->   String))
--             [1]       [2]       [3]
```

1. First, outermost `(->)` type constructor. This has the type `a -> (b -> (c -> String))`. So the argument is `a` and the result is `b -> (c -> String)`.

2. Second function, nested inside the first function, has the type `b -> (c -> String)`, so the argument is `b` and the result is `c -> String`.

3. Third, innermost function. Has the type `c -> String`, so the argument is `c` and the result is `String`.

Manual currying and Uncurry

Haskell is curried by default, but you can uncurry functions. "Uncurrying" means un-nesting the functions and replacing the two functions with a tuple of two values (these would be the two values you want to use as arguments). If you uncurry `(+)`, the type changes from `Num a => a -> a -> a` to `Num a => (a, a) -> a` which better fits the description "takes two arguments, returns one result" than curried functions. Some older functional languages default to using a product type like tuples to express multiple arguments.

Uncurried functions	Curried functions
One function, many arguments	Many functions, one argument apiece

You can also de-sugar the automatic currying yourself, by nesting the arguments with lambdas, though there's almost never a reason to do so.

We've used some syntax here that we haven't introduced yet. This anonymous lambda syntax will be fully explained in a later chapter. For now, you can get a sense of what's happening here without knowing that bit of syntax by comparing the functions directly and thinking of the backslash as a lambda:

```haskell
nonsense :: Bool -> Integer
nonsense True = 805
nonsense False = 31337

typicalCurriedFunction :: Integer -> Bool -> Integer
typicalCurriedFunction i b = i + (nonsense b)

anonymous :: Integer -> Bool -> Integer
anonymous = \i b -> i + (nonsense b)

anonymousAndManuallyNested :: Integer -> Bool -> Integer
anonymousAndManuallyNested = \i -> \b -> i + (nonsense b)
```

Then when we test the functions from the REPL:

```
Prelude> typicalCurriedFunction 10 False
31347
Prelude> anonymous 10 False
31347
Prelude> anonymousAndManuallyNested 10 False
31347
```

They are all the same function, all giving the same results. In "anonymousAndManuallyNested", we just manually nested the anonymous lambdas to get a function that was semantically identical to "typicalCurriedFunction" but didn't leverage the automatic currying. This means functions that *seem* to accept multiple arguments such as with `a -> a -> a -> a` are really *higher-order functions:* they yield more function values as each argument is applied until there are no more `(->)` type constructors and it terminates in a non-function value.

Intermission: Exercises

Given a function and its type, tell us what type results from applying some or all of the arguments.

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`

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"`

   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"`

   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`

To review your answers for the above, you can do this in the REPL, changing the relevant information for each:

```
Prelude> let f :: a -> a -> a -> a; f = undefined
Prelude> :t f undefined
f undefined :: a -> a -> a
Prelude> :t f 1
f 1 :: Num a => a -> a -> a
Prelude> :t f (1 :: Int)
f (1 :: Int) :: Int -> Int -> Int
```

5.6 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 with *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 the numerical functions that come with your GHC installation and the Haskell 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 typeclasses.

Parametric polymorphism is broader than ad-hoc polymorphism. Parametric polymorphism refers to type variables, or *parameters*, that are fully polymorphic. When unconstrained by a typeclass, their final, concrete type could be anything. Constrained polymorphism, on the other hand, puts typeclass constraints on the variable, decreasing the number of concrete types it could be, but increasing what you can actually do with it by defining and bringing into scope a set of operations.

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 Haskell 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 typeclass. Passing any value to `id` will return the same value:

```haskell
id :: a -> a
-- For all 'a', get an argument of some type 'a',
-- return value of 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
```

---

[2]Wadler's paper on making Ad-hoc polymorphism less ad-hoc `http://people.csail.mit.edu/dnj/teaching/6898/papers/wadler88.pdf`

```
1
Prelude> id "blah"
"blah"
Prelude> let inc = (+1)
Prelude> inc 2
3
Prelude> (id inc) 2
3
```

Based on the type of **id**, we are guaranteed this 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 (exceptions will be discussed later), 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 typeclass, so the terms of the function are more restricted because there are no methods or information attached to them. With the type **id :: a -> a**, it can do nothing other than return **a** because there is no information or method attached to its argument at all — nothing can be done *with* **a**. On the other hand, a function like **negate**, with a similar-appearing type signature of **Num a => a -> a** constrains the **a** variable as an instance of the **Num** typeclass. Now **a** has fewer concrete types it could be, but there are a set of methods you can use, a set of things that can be done with **a**.

If a type is a set of possible values, then a type variable represents a set of possible types. When there is no typeclass constraint, the set of possible types a variable could represent is effectively unlimited. Typeclass 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 typeclasses for one thing. For example, an **Int** is only an **Int**, but it can make use of the methods of the **Num** *and* **Integral** typeclasses because it has instances of both.

In sum, if a variable could be *anything*, then there's little that can be done to it because it has no methods. If it can be *some* types (say, a type that is 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 typeclass 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 is an instance of `Num` but not an instance of `Integral`, it can't implement the methods of the `Integral` typeclass. 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 typeclass) parameters. Parametricity is the property we get from having parametric polymorphism. *Parametricity* means that the behavior of a function with respect to the types of its (parametrically polymorphic) arguments is uniform. The behavior can *not* change just because it was applied to an argument of a different type.

Intermission: Exercises

All you can really 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 it do something other than returning 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 two– and only two–implementations. Write both possible versions of `a -> a -> a`, then try to violate the constraints we've described.

3. Implement `a -> b -> b`. How many implementations can it have? Does the 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 they cannot be used willy-nilly wherever we like; there are constraints on using different types of numbers in different functions. But intuitively we 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 stay so until given a more specific type. The `Num a =>` or `Fractional a =>` is a typeclass constraint and the `a` is the type variable in scope. In the type for the entire equation, we see that the compiler inferred that it was working with `Fractional` numbers. It had to, 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` typeclass, which could be any type of number. 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.

We can force the compiler to be more specific about the types of numbers by declaring the type:

```
Prelude> let x = 5 + 5
Prelude> :t x
x :: Num a => a
```

```
Prelude> let x = 5 + 5 :: Int
Prelude> :t x
x :: Int
```

In the first example, we did not specify a type for the numbers, so the type signature defaulted to the broadest interpretation, but in the second version, we told the compiler to use the `Int` type.

Working around constraints

Previously, we've 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 `length` isn't polymorphic enough. `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` typeclass, 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> 6 / fromIntegral (length [1, 2, 3])
2.0
```

And now all is right with the world once again.

5.7   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 whose 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 was 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> let 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. 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> let 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, see what happens:

```
Prelude> let 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:

```haskell
-- typeInference1.hs
module TypeInference1 where

f :: Num a => a -> a -> a
f x y = x + y + 3
```

Then when we load the code into GHCi to experiment:

```
Prelude> :l typeInference1.hs
[1 of 1] Compiling TypeInference1
Ok, modules loaded: TypeInference1.
```

```
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 (typeclass 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, modules loaded: TypeInference2.
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 typeclasses in a way that doesn't make it clear which type you mean unless you assert one.

Intermission: Exercises

Look at these pairs of functions. One function is unapplied, so the compiler will infer maximally polymorphic type. The second function has been ap-

plied 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 would 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'
   ```

5.8 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've seen before. If we allow the compiler to infer the type, we end up with this:

```
Prelude> let triple x = x * 3
Prelude> :type triple
triple :: Num a => a -> a
```

Here the triple function was made from the `(*)` function which has type `(*) :: Num a => a -> a -> a`, but we have already applied one of the arguments, which is the 3, so there is one less 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 may only be integers, this is how we declare that:

```
Prelude> let triple x = x * 3 :: Integer
Prelude> :t triple
triple :: Integer -> Integer

-- Note the typeclass 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 more like what you would see in a source file:

```
-- declaration of triple's type
triple :: Integer -> Integer

-- declaration of the function
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. The `let` and `where` clauses we've seen before are for declarations within other expressions.

It is possible, though uncommon, to declare types locally with `let` and `where` clauses. Here's an example of assigning a type within a `where` clause:

```
triple x = tripleItYo x
  where tripleItYo :: Integer -> Integer
        tripleItYo y = y * 3
```

Also note that we don't have to assert the type of `triple`:

```
Prelude> :t triple
triple :: Integer -> Integer
```

The assertion in the `where` clause narrowed our type down from `Num a => a -> a` to `Integer -> Integer`. It's worth remembering that 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. For example, if we try to make the `+` function return a `String`, we get an error message:

```
Prelude> let 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 is just telling us that 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` typeclass 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` typeclass.

5.9 Chapter Exercises

Multiple choice

1. A value of type `[a]` is

   a) a list of alphabetic characters

   b) a list of lists

   c) a list whose elements are all of some type `a`

   d) a list whose elements 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 in to 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` be of different types

Determine the type

For the following functions, determine the type of the specified value. Note: you can type them into a file and load the contents of the file in GHCi. You can then query the types after you've loaded them.

1. All function applications return a value. Determine the value returned by these function applications and the type of that value.

    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

```haskell
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 (n.b. we do not mean literal squawking) and why. Fix it if you can.

1. ```haskell
   bigNum = (^) 5 $ 10
   wahoo = bigNum $ 10
   ```

2. ```haskell
   x = print
   y = princ "woohoo!"
   z = x "hello world"
   ```

3. ```haskell
   a = (+)
   b = 5
   c = b 10
   d = c 200
   ```

4. ```haskell
   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, constrained polymorphic type variable, or concrete type constructor.

```
f :: Num a => a -> b -> Int -> Int
--              [0]  [1]   [2]    [3]
```

Here, the answer would be: constrained polymorphic (Num), fully polymorphic, concrete, and concrete.

2. Categorize each component of the type signature as described in the previous example.

```
f :: zed -> Zed -> Blah
```

3. Categorize each component of the type signature

```
f :: Enum b => a -> b -> C
```

4. Categorize each component of the 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've seen 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.

```
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).

1. There is only one implementation that typechecks.

   ```
   i :: a -> a
   i = undefined
   ```

2. There is only one version that works.

   ```
   c :: a -> b -> a
   c = undefined
   ```

3. Given alpha equivalence are c" and c (see above) the same thing?

   ```
   c'' :: b -> a -> b
   c'' = ?
   ```

4. Only one version that 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 that will typecheck.

   ```
   co :: (b -> c) -> (a -> b) -> (a -> c)
   co = undefined
   ```

7. One version will typecheck.

```haskell
a :: (a -> c) -> a -> a
a = undefined
```

8. One version will typecheck.

```haskell
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.
```haskell
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 and make it sing the other song. If you're lucky, you'll end up with both songs stuck in your head!

3.
```
-- 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 Phillip Wright[3], 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 the practice will make it easier to deal with when you get there. Practicing this 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 typechecker by modifying it alone.

Here's a worked example for how we present these exercises and how you are expected to solve them. Given the following:

---

[3]https://twitter.com/SixBitProxyWax

```haskell
data Woot

data Blah

f :: Woot -> Blah
f = undefined

g :: (Blah, Woot) -> (Blah, Blah)
g = ???
```

Here it's **g** that you're supposed to implement, however you can't evaluate anything. You're to only use type-checking and type-inference to validate your answers. Also note that we're using a trick for defining datatypes which can be named in a type signature, but have no values. Here's an example of a valid solution:

```haskell
g :: (Blah, Woot) -> (Blah, Blah)
g (b, w) = (b, f w)
```

The idea is to only fill in what we've marked with **???**.

Not all terms will always be used in the intended solution for a problem.

1. ```haskell
   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 = ???
   ```

5.10   Definitions

1. *Polymorphism* refers to type variables which 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 the commonalities of them all to produce a smaller set of correct terms. This makes it less likely we'll write an incorrect program and lets us reuse the code with other types.

2. With respect to Haskell, the *principal type* is the most generic type which still typechecks. More generally, *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. Here are some examples:

```haskell
-- 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. *Type inference* is a faculty some programming languages, most notably Haskell and ML, have to *infer* principal types from terms without needing explicit type annotations. There are, in some cases, terms in Haskell which can be well-typed but which have no principal type. In those cases, an explicit type annotation must be added.

4. *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

-- One type variable 'a' that occurs twice,
-- once as an argument, once as a result.
-- Parametrically polymorphic, could be
-- strictly anything

(+) :: Num a => a -> a -> a

-- One type variable 'a', constrained to needing
-- an instance of Num. Two arguments, one result.
-- All the same type.
```

5. A *Typeclass* is a means of expressing faculties or interfaces that multiple datatypes may have in common and then write code just in terms of those commonalities without repeating yourself 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 by unifying them into a single typeclass. Importantly, these can then be used with *all* types that have a `Num` instance. Thus, a typeclass 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 typeclass, whether they already exist or are yet to be invented (by you, perhaps).

6. *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 the function. Parametricity[4] tells us that the function:

```
id :: a -> a
```

---

[4]Examples are courtesy of the `@parametricity` twitter account.
`https://twitter.com/parametricity`

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:

```
const :: a -> b -> a
```

`const` *must* return the first value — parametricity and the definition of the type requires it!

```
f :: a -> a -> a
```

`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 typeclass `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` which requires a throw-away argument for no reason.

```
convList :: Maybe a -> [a]
```

Unless the result is `[]`, the resulting list has values that are all the same value.

7. *Ad-hoc polymorphism* (sometimes called "constrained polymorphism") is polymorphism that applies one or more typeclass 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 typeclass that defines the methods and Haskell's requirement that typeclass instances be unique for a given type. For any given combination of typeclass and a type, such as `Ord` and `Bool`, there must only exist one unique instance in scope. This makes it considerably easier to reason about typeclasses. See the example for a disambiguation.

```
(+) :: Num a => a -> a -> a

-- the above function is leveraging
-- ad-hoc polymorphism via the Num typeclass

c' :: a -> a -> a

-- This function is not,
-- it's parametrically polymorphic in 'a'.
```

5.11   Follow-up resources

1. Luis Damas; Robin Milner. Principal type-schemes for functional programs
   `http://web.cs.wpi.edu/~cs4536/c12/milner-damas_principal_types.pdf`

2. Christopher Strachey. Fundamental Concepts in Programming Languages
   `http://www.cs.cmu.edu/~crary/819-f09/Strachey67.pdf`
   Popular origin of the parametric/ad-hoc polymorphism distinction.

Chapter 6

Typeclasses

Slightly less ad-hoc polymorphism

A blank cheque kills creativity.

Mokokoma Mokhonoana

6.1 Typeclasses

You may have realized that it is very difficult to talk about or understand Haskell's type system without also talking about typeclasses. So far we've been focused on the way they interact with type variables and numeric types, especially. This chapter explains some important predefined typeclasses, only some of which have to do with numbers, as well as going into more detail about how typeclasses work more generally. In this chapter we will

- examine the typeclasses Eq, Num, Ord, Enum, and Show;

- learn about type-defaulting typeclasses and typeclass inheritance;

- look at some common but often implicit functions that create side effects.

6.2 What are typeclasses?

Typeclasses and types in Haskell are, in a sense, opposites. Where a declaration of a type defines how that type in particular is created, a declaration of a typeclass 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 typeclasses as being like *interfaces* to data that can work across multiple datatypes. The latter facility is why typeclasses are a means of ad-hoc polymorphism — "ad-hoc" because typeclass 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 more clear.

---

[1]Philip Wadler, "The Expression Problem" `http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt`

Simply put, typeclasses allow us to generalize over a set of types in order to define and execute a standard set of features for those types. For example, the ability to test values for equality is useful, and we'd want to be able to use that function for data of various types. In fact, we can test any data that has a type that implements the typeclass 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 typeclass, we can use the standard functions. Also, all the numeric literals and their various types implement a typeclass called Num, which defines a standard set of operators that can be used with any type of numbers.

6.3   Back to Bool

Let's return briefly to the Bool type to get a feel for what typeclass information really looks like. As you may recall, we can use the GHCi command `:info` to query information, including typeclass information about any function or type (and some values):

```
Prelude> :info Bool
data Bool = False | True
instance Bounded Bool
instance Enum Bool
instance Eq Bool
instance Ord Bool
instance Read Bool
instance Show Bool
```

The information includes the data declaration for Bool. In your REPL, it also tells you where the datatype and its instances are defined for the compiler. Next we see a list of instances. Each of these instances is a typeclass that Bool implements, and the instances are the unique specifications of how Bool makes use of the methods from that typeclass. 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 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.

Typeclasses have a hierarchy of sorts,[2] as you might have guessed from our discussion of numeric types. All `Fractional` numbers implement the `Num` typeclass, but not all `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 something in an enumerated list, they must be able to be ordered; to be able to order something, they must be able to be compared for equality.

6.4   Eq

In Haskell, equality is implemented with a typeclass called Eq. Some programming languages bake equality into every object in the language, but some datatypes do not have a sensible notion of equality, 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 at `http://haskell.org/hoogle` to find information on Haskell datatypes and typeclasses. This is useful for looking up what the deal is with things like Fractional. Hoogle is a Haskell API search engine, which allows you to search many standard Haskell libraries by function name or type signature. This becomes very useful as you become fluent in Haskell types as you will be able to input the type of the function you want and find the functions that match.

```
Prelude> :info Eq
class Eq a where
(==) :: Eq a => a -> a -> Bool
(/=) :: Eq a => a -> a -> Bool
-- List of some Eq instances
instance Eq a => Eq [a]
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 Integer
```

If you do this in your REPL, you'll actually see *a lot* more information than this, but let's focus on this. First, it tells us we have a typeclass called Eq where there are two basic functions, equality and nonequality, and gives their type signatures. Next we see some of the instances of Eq. 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 are using data of these types, we are implementing the Eq typeclass and therefore have generic functions we can use to compare their equality. Any type that has an instance of this typeclass implements the methods of the typeclass.

Here are some examples using this typeclass:

```
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 which implements the Eq typeclass. We also know that both functions will take two arguments of the same type a and return 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 specialized (==) 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 first two arguments a and a aren't the same type?

```
Prelude F M> (1, 2) == "puppies!"

<interactive>:259:11:
    Couldn't match expected type '(t0, t1)' 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 '(t0, t1)'
  with actual type '[Char]'
```

This error means our [Char] wasn't the 2-tuple of types $t0$ and $t1$ that was expected. `(t0, t1)` was expected for the second argument where we supplied `"puppies!"` because that's the type of the first argument. Remember: the type of a is usually set by the leftmost instance of it and can't change in the signature `Eq a => a -> a -> Bool`. Applying `(==)` to Integer will bind the a type variable to Integer. This is as if the signature changed to `Eq Integer => Integer -> Integer -> Bool`. The typeclass constraint `Eq Integer =>` gets dropped because it's redundant.

We can see the issue more clearly if we look at the typeclass instances on the 2-tuple `(,)`:

```
data (,) a b = (,) a b
instance (Eq a, Eq b) => Eq (a, b)
instance (Ord a, Ord b) => Ord (a, b)
instance (Read a, Read b) => Read (a, b)
instance (Show a, Show b) => Show (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 the equality of two tuples `(a, b)` depends on the equality of their constitutent values a and b. This 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)
```

6.5 Num

Num is a typeclass implemented by most numeric types. As we see when
we query the information, it has a set of predefined functions, like Eq:

```
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
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 typeclass, numeric types
we've looked at previously. No surprises here.

Integral

The typeclass 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 typeclass constraint `(Real a, Enum a) =>` means that any type that implements Integral must already have instances for Real *and* Enum typeclasses. In a very real sense the tuple syntax here denotes the conjunction of typeclass constraints on your type variables. An integral type must be both a real number and enumerable (more on Enum later) and therefore may employ the methods each of those typeclasses. In turn, the Real typeclass itself requires an instance of Num. So, the Integral typeclass 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 typeclass inheritance is *only* additive and the ambiguity problems caused by multiple inheritance in some programming languages — the so-called "deadly diamond of death" — are avoided.

Intermission: Exercises   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> let ones x = snd (divMod x 10)
```

Fractional

`Fractional` is an instance of Num. The Fractional typeclass is defined as follows:

```haskell
class  (Num a) => Fractional a  where
    (/)          :: a -> a -> a
    recip        :: a -> a
    fromRational :: Rational -> a
```

This typeclass 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 typeclass 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 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 the chapter to help you visualize this information.

We can see this just with ordinary functions:

First let us consider the function, intentionally without a type provided:

```haskell
divideThenAdd x y = (x / y) + 1
```

We'll load this with a type that asks only for a Num instance:

```haskell
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 :: 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:

```
-- This works fine.
-- (+) and (-) are both provided by Num

subtractThenAdd :: Num a => a -> a -> a
subtractThenAdd x y = (x - y) + 1
```

Or we can change the type rather than modifying the function itself:

```
-- This works fine.

divideThenAdd :: Fractional a => a -> a -> a
divideThenAdd x y = (x / y) + 1
```

Put on your thinking cap Why didn't we need to make the type of the function we wrote require both typeclasses? 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.

6.6 Type-defaulting typeclasses

When you have a typeclass-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 typeclass 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 typeclass 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[3] 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 typeclasses, and Integer and Double are the types
they default to. This type defaulting for Fractional means that `(/) ::`

---

[3]The Haskell Report is the standard 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/`.

`Fractional a => a -> a -> a` changes to `(/) :: Double -> Double -> Double` if you don't specify the type. A similar example but for Integral would be `div :: Integral a => a -> a -> a` defaulting to `div :: Integer -> Integer -> Integer`. The typeclass constraint is superfluous when the types are concrete.

On the other hand, you must specify which typeclasses you want type variables to have implemented. The use of polymorphic values without the ability to infer a specific type and no default rule will cause GHC to complain about an ambiguous type.

The following will work because all the types below implement the Num typeclass:

```
Prelude> let x = 5 + 5 :: Int
Prelude> x
10

Prelude> let x = 5 + 5 :: Integer
Prelude> x
10

Prelude> let x = 5 + 5 :: Float
Prelude> x
10.0

Prelude> let 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
```

Now 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 Integer
(+)   :: Integer -> Integer -> Integer
-- Apply the first argument
(x+)  ::            Integer -> Integer
-- Applying the second and last argument
(x+y) ::                       Integer
-- Final result was Integer.
```

We can declare more specific (monomorphic) functions from more general (polymorphic) functions:

```
let addIntegers = (+) :: Integer -> Integer -> Integer
```

We cannot go in the other direction, because we lost the generality of Num when we specialized to Integer:

```
Prelude> :t id
id :: a -> a
Prelude> let numId = id :: Num a => a -> a
Prelude> let intId = numId :: Integer -> Integer
Prelude> let altNumId = intId :: Num a => a -> a
```

```
Could not deduce (a1 ~ Integer)
from the context (Num a)
  bound by the inferred type of
     altNumId :: Num a => a -> a

or from (Num a1)
  bound by an expression type signature:
    Num a1 => a1 -> a1

  'a1' is a rigid type variable bound by
      an expression type signature:
        Num a1 => a1 -> a1

Expected type: a1 -> a1
  Actual type: Integer -> Integer
In the expression: intId :: Num a => a -> a

In an equation for 'altNumId':
  altNumId = intId :: Num a => a -> a
```

The "Expected type" and the "Actual type" don't match, and the actual type is more concrete than the expected type. Types can be made more specific, but not more general or polymorphic.

6.7 Ord

Next we'll take a look at a typeclass called `Ord`. We've previously noted that this typeclass 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 typeclass. 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 (Maybe a)
instance (Ord a, Ord b) => Ord (Either a b)
instance Ord Integer
instance Ord a => Ord [a]
instance Ord Ordering
instance Ord Int
instance Ord Float
instance Ord Double
instance Ord Char
instance Ord Bool
```

Notably, at the top, we have another typeclass constraint. Ord is constrained by Eq because if you're going to compare items in a list and put them in order, you have to have a way to discover if they are equal in value included in that. So, Ord implements 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 typeclass, 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 typeclass:

```
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 the type signature, we can see that these functions will accept two parameters. If you want to use these 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 parameters, you will get this strange-seeeming message:

```
Prelude> max "Julie"
```

```
No instance for (Show ([Char] -> [Char]))
--     [1]         [2]   [     3     ]
arising from a use of 'print'
--                        [4]
In a stmt of an interactive GHCi command: print it
--              [              5              ]
```

1. Haskell couldn't find an instance of a typeclass for a value of a given type

2. The typeclass it couldn't find an instance for was `Show`, the typeclass that allows GHCi to print values in your terminal. More on this in the following sections.

3. It couldn't find an instance of Show for the type `String -> String`. Nothing with `(->)` should have a Show instance as a general rule because `(->)` denotes a function rather than a constant value.

4. We wanted an instance of Show because we (indirectly) invoked `print` which has type `print :: Show a => a -> IO ()` - note the constraint for Show.

5. The interactive GHCi command "print it" invoked `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 is 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.

Intermission: Exercises

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 answer):

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)`

6.8   Enum

A typeclass known as `Enum` that we have mentioned previously seems similar to Ord but is slightly different. This typeclass covers types that are enumerable, therefore have known predecessors and successors. We shall try not to belabor the point, because you are probably developing a good idea of how to query and make use of typeclass 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 Ordering
```

```
instance Enum Integer
instance Enum Int
instance Enum Char
instance Enum Bool
instance Enum ()
instance Enum Float
instance Enum Double
```

Remember the "instances" are a list of different types that implement the methods of this typeclass: numbers and characters are certainly known to 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'`?

6.9 Show

`Show` is a typeclass that provides for the creating of human-readable string representations of structured data. GHCi uses Show to create 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 typeclass information looks like this (truncated, because it also has instances for many tuple types which becomes very lengthy):

```haskell
class Show a where
  showsPrec :: Int -> a -> ShowS
  show :: a -> String
  showList :: [a] -> ShowS

instance Show a => Show [a]
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 built-in ability to be printed to the screen. There is also a function `show` which 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 the chapter about printing and again in the section about Ord and the error message that results from passing the `max` function too few arguments. As understanding `print` is important to understanding this typeclass, 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 without needing to resort to something more to write a Haskell program.

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 covering over how this works, but it's time to dive a bit deeper.

`print` is defined in the Haskell Prelude standard as a function to output "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 next look at the type of `print`:

```
Prelude> :t print
print :: Show a => a -> IO ()
```

As we see, `print` takes an argument *a* that is an instance of the Show typeclass and returns an `IO ()` result. This result is an `IO` action that returns a value of the type `()`.

We saw this `IO ()` result previously when we talked about printing strings. We also noted that it is the obligatory type of the `main` function in a source code file. This is because running `main` *only* produces side effects.

Stated as simply as possible, an IO (input/output, frequently written 'IO' without a slash) 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, that 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 IO 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 with `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.

```haskell
-- this is just a String value
myVal :: String

-- this value is a method or means of obtaining
-- a value of type String which
-- performs side effects aka IO
ioString :: IO String
```

An `IO` action is performed when we run the `main` function of our program, as we have seen. But we also perform an `IO` action when we invoke `print` implicitly or explicitly.

Working with Show

We haven't discussed it much yet, but it is possible to define our own datatypes (as well as our own typeclasses). That will be covered in depth in a later chapter, but for now we will look at some trivial examples of doing so in order to illustrate why Show is important and how it is implemented.

A minimal implementation of an instance of Show only requires that `show` or `showsPrec` be implemented, as in the following example:

```haskell
data Mood = Blah

instance Show Mood where
  show _ = "Blah"


*Main> 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 typeclass:

```
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 and should just derive the Show instance for Mood because it's one of the typeclasses GHC supports deriving instances for by default:

```
Prelude> data Mood = Blah deriving Show
Prelude> Blah
Blah
```

Invoking the Show typeclass also invokes its methods, specifically a method of taking your values and turning them into values that can be printed to the screen.

Typeclass deriving Typeclass instances we can magically derive are `Eq`, `Ord`, `Enum`, `Bounded`, `Read`, and `Show`, though there are some constraints on deriving some of these. Deriving means you don't have to manually write instances of these typeclasses for each new datatype you create. We'll address this a bit more in the chapter on Algebraic Datatypes.

6.10   Read

The `Read` typeclass...well, it's...*there.* You'll notice that, like Show, a lot of types have instances of Read. This typeclass is essentially the opposite of Show. Where 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 are in no way guaranteed 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.

6.11 Typeclass instances are dispatched by type

We've said a few times, without explaining it, that typeclasses are dispatched by type, but it's an important thing to understand. Typeclasses are defined by the set of operations and values all instances will provide. The typeclass *instances* are unique pairings of the typeclass an instance is being defined for and the type it's for.

We're going to walk through some code to illustrate what this all means. The first thing you will see is that we've written our own typeclass and

instances for demonstration purposes. We will talk about how to write your own typeclasses later in the book. Those details aren't important for understanding this code. Just remember:

- a typeclass defines a set of functions and/or values
- types have instances of that typeclass
- the instances specify the ways that type uses the functions of the typeclass

This is vacuous and silly. This is only to make a point. Please do not write typeclasses like this:

```haskell
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 typeclass 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
```

Now let us think about this for a moment. The class definition of Numberish doesn't define any *terms* or actual code we can compile and execute, only types. The code actually lives in the instances for Age and Year. So how does Haskell know where to find code?

```
Prelude> sumNumberish (Age 10) (Age 10)
Age 20
```

In the above, it knew to use the instance of Numberish for Age because it could see that our arguments to `sumNumberish` were 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 argument is applied to a value of type Age, it knows that all other occurrences of type `Numberish a => a` must actually 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 typeclass and associated instances:

```haskell
-- This is even worse than the last one.
-- Don't use typeclasses 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

No instance for (Show a0) arising from a use of 'print'
The type variable 'a0' is ambiguous
 Note: there are several potential instances:
  instance Show a => Show (Maybe a) -- Defined in 'GHC.Show'
  instance Show Ordering -- Defined in 'GHC.Show'
  instance Show Integer -- Defined in 'GHC.Show'
      ...plus 24 others
```

This fails because it has *no idea* what type `defaultNumber` is other than that it's provided for by Numberish's instances. 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, what typeclass instance we want to get our `defaultNumber` from.

Why not write a typeclass like this?

For reasons we'll explain when we talk about Monoid, it's important that your typeclasses have laws and rules about how they work. Numberish is a bit arbitrary, and there are better ways to express what it does in Haskell than a typeclass. Functions and values alone suffice here.

6.12   Writing instances for your typeclasses

If you've written your own typeclass, as we did with Numberish above, you need to write your own instances for different datatypes. You may also sometimes need to write instances of existing typeclasses for datatypes you've written (we'll talk more about this later). We'll focus on the Eq typeclass to demonstrate how to do this.

Eq instances

As we've seen, Eq provides instances for determining equality of values and may be the simplest and most common typeclass you'll want to use with your own datatypes. You can investigate a typeclass by referring to the Hackage documentation for that typeclass. Typeclasses 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 `(==)` (equal) or `(/=)` (unequal) 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 just 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 butt of this datatype declaration, we'll have no typeclass 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 (Show / print) or equality (Eq (value equality) or pointer 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:

```haskell
-- keep your typeclass 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! If you load this up, you have only one possible expression you can construct here:

```haskell
Prelude> Trivial' == Trivial'
True
```

Let's drill down a bit into how this instance stuff works:

```haskell
  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 typeclass instance. Typeclass instances are how you tell Haskell how equality, stringification (Show), orderability (Ord), enumeration (Enum) or other typeclasses 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 typeclass 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 typeclass *for* the Trivial datatype.

4. The keyword `where` terminates the initial declaration and beginning of the instance. What follows are the actual 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 `==`, this 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 `==`.

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 just being shown as an alterative; you can't have two typeclass instances for the same type. We'll explain more about this later, but typeclass instances are unique to a given type. You can try having both in the same file, but you'll get an error.

Okay, let's stretch our legs a bit and try something a bit less...Trivial!

```
data DayOfWeek =
  Mon | Tue | Weds | Thu | Fri | Sat | Sun

-- day of week and numerical day of month
data Date =
  Date DayOfWeek Int

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

instance Eq Date where
  (==) (Date weekday monthNum)
       (Date weekday' monthNum') =
    weekday == weekday' && monthNum == monthNum'
```

One thing to particularly note here is that in the Eq instance for Date, we didn't recapitulate how equality for DayOfWeek and Int values worked; we simply said that the dates were equal if all of their constitutent values were equal.

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!

Partial functions — not so strange danger

We need to take care to avoid partial functions in general in Haskell, but this must be especially kept in mind when we have a type with multiple "cases" such as DayOfWeek. What if we had made a mistake in the Eq instance?

```haskell
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 sucks. We definitely didn't start learning Haskell because we wanted 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 (such as with Cabal — more on that later), 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

code/derivingInstances.hs:19:3: Warning:
    Pattern match(es) are non-exhaustive
    In an equation for '==':
        Patterns not matched:
            Mon Tue
            Mon Weds
            Mon Thu
            Mon Fri
            ...
Ok, modules loaded: DerivingInstances.
```

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.

This doesn't only work with user-defined sum types either. It also works for types like Int:

```
-- this'll blow up for any input not ``1''

f :: Int -> Bool
f 1 = True
```

If you compile or load this, you'll get another warning. 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 1:

```
Pattern match(es) are non-exhaustive
In an equation for 'f':
  Patterns not matched: GHC.Types.I# #x with #x `notElem` [1#]
```

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:

```haskell
f :: Int -> Bool
f 1 = True
f 2 = True
```

```
Pattern match(es) are non-exhaustive
In an equation for 'f':
  Patterns not matched: GHC.Types.I# #x with #x `notElem` [1#, 2#]
```

```haskell
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:
    GHC.Types.I# #x with #x `notElem` [1#, 2#, 3#]
```

So on and so forth. The real answer here is to have an unconditional case
that matches everything:

```haskell
-- This 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 effin' huge like Int if you
only have a few cases you want to consider.

```haskell
-- Seriously. It's huge.
```

```
Prelude> minBound :: Int
-9223372036854775808
Prelude> maxBound :: Int
9223372036854775807
```

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 typeclass such as Eq for something that takes a polymorphic argument, such as Identity below, we'll sometimes need to require our argument or arguments to provide some typeclass 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 (written as a in the datatype declaration and v in the instance definition) 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 really know anything about a. In this case, we can't assume it has an Eq instance. However, we can use the same typeclass constraint syntax we saw with functions, in our instance declaration:

```haskell
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.

```
Prelude> Identity NoEqInstance == Identity NoEqInstance

No instance for (Eq NoEqInstance)
arising from a use of '=='

In the expression:
  Identity NoEqInstance == Identity NoEqInstance

In an equation for 'it':
  it = Identity NoEqInstance == Identity NoEqInstance
```

We could actually 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:

```haskell
instance Ord a => Eq (Identity a) where
  (==) (Identity v) (Identity v') =
    compare v v' == EQ
```

That will compile, but it's not clear why you'd do it. Maybe you have your own secret reasons.

Intermission: Exercises

Write the Eq instance for the datatype 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
   ```

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 did want to point out here that 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:

```haskell
data DayOfWeek =
  Mon | Tue | Weds | Thu | Fri | Sat | Sun
  deriving (Ord, Show)
```

We only derived 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:

```
*Main> Mon > Tue
False
*Main> Sun > Mon
True
*Main> compare Tue Weds
LT
```

But if we wanted to express that Friday is always the best day, we can write our own Ord instance to express that:

```haskell
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, you notice, are equal in value:

```
*Main> compare Fri Sat
GT
*Main> compare Sat Mon
EQ
*Main> compare Fri Mon
GT
*Main> compare Sat Fri
LT
*Main> Mon > Fri
False
*Main> Fri > Sat
True
```

But we did derive an Eq instance above, so we do get the expected equality behavior:

```
*Main> Sat == Mon
False
*Main> 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`.

We'll look into writing our own typeclasses and typeclass instances more, later in the book.

6.13 Gimme more operations

When we talked about the different kinds of polymorphism in type signatures — constrained versus 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 typeclasses, and so we get more utility by specifying typeclass constraints. If your types are more general than your terms are, then you need to constrain your types with the typeclasses 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 times when 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 :: a -> a -> a
```

Fortunately, this is one of those cases where GHC knows precisely what the problem is and how to remedy it. We just 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, so long as we resist
the pull of tunnel vision[4] and 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 :: Num a => a -> a -> a

  Possible fix:
    add (Ord a) to the context of
      the type signature for addWeird :: Num a => a -> a -> a
```

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 told us to do:

```
addWeird :: (Ord a, Num a) => a -> a -> a
addWeird x y =
  if x > 1
  then x + y
  else x
```

Now this should typecheck because our constraints are asking that *a* have
instances of Num *and* Ord.

---

[4]All programmers experience this. Just slow down and you'll be okay.

Ord implies Eq

The following isn't going to typecheck for reasons we already covered:

```haskell
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' :: 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?

```haskell
check' :: Ord a => a -> a -> Bool
check' a a' = a == a'
```

It should compile just fine. Now, Ord isn't what GHC asked for, so why did it work? It worked 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* which 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 was "real" code — but we did this so you could get an idea of how constraints and superclassing in Haskell work.

Concrete types imply all the typeclasses they provide

Let's take the last few examples and modify them to all have a concrete type in the place of *a*:

```haskell
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 typecheck! This is because the Int type has the typeclasses 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 typeclass instance or it doesn't — adding the constraint means nothing. A concrete type always implies the typeclasses 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 typeclasses 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 typeclasses and operations defined for it — it could be easy to make a function that does something unintended. Whereas if we were to write a function, even if we had Int values in mind for it, which used

a polymorphic type constrained by the typeclass instances we wanted, we could ensure we only used the operations we intended. This isn't a panacea, but it can be worth avoiding concrete types for these (and other) reasons sometimes.

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 typeclass 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 typeclass 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 typeclass Integral includes

   a) Int and Integer numbers

   b) integral, real, and fractional numbers

   c) Schrodinger's cat

   d) only positive numbers

Does it typecheck?

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 typeclass that lets you convert things to Strings (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'

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 it can't find an implementation of the typeclass Show for the type `Int -> Int`, which makes sense. Nothing with the function type constructor `(->)` has an instance of Show[5] by default in Haskell.

Examine the following code and decide whether it will typecheck. Then load it in GHCi and see if you were correct. If it doesn't typecheck, 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 typecheck? If not, why not?

   ```haskell
   data Person = Person Bool

   printPerson :: Person -> IO ()
   printPerson person = putStrLn (show person)
   ```

2. Does the following typecheck? If not, why not?

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

   settleDown x = if x == Woot
                     then Blah
                     else x
   ```

3. If you were able to get `settleDown` to typecheck:

   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 typecheck? If not, why not?

---

[5]For an explanation and justification of why functions in Haskell cannot have a Show instance, see the wiki page on this topic. `https://wiki.haskell.org/Show_instance_for_functions`

```
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 typecheck? For the ones that don't typecheck, 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* just 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, then tested that answer and believe you understand why you were right or wrong, make sure to use GHCi to check what type GHC *infers* for the definitions we provide without a type assigned. For example, for this one, you'd type in:

      ```
      Prelude> let i = 1
      Prelude> :t i
      -- Result elided intentionally.
      ```

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

Round Two! Same rules apply — you're trying to fill in terms (code) which'll fit the type. The idea with these exercises is that you'll derive the implementation from the type information. You'll probably need to use stuff from Prelude.

1.
```haskell
chk :: Eq b => (a -> b) -> a -> b -> Bool
chk = ???
```

2.
```haskell
-- Hint: use some arithmetic operation to
-- combine values of type 'b'. Pick one.
arith :: Num b => (a -> b) -> Integer -> a -> b
arith = ???
```

6.15 Chapter Definitions

1. *Typeclass inheritance* is when a typeclass has a superclass. This is a way of expressing that a typeclass requires *another* typeclass to be available for a given type before you can write an instance.

   ```haskell
   class Num a => Fractional a where
     (/) :: a -> a -> a
     recip :: a -> a
     fromRational :: Rational -> a
   ```

   Here the typeclass 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.

   ```haskell
   -- Even though in principle
   -- this 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. *Side 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 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 typeclass should work for a given type. Instances are unique for a given combination of typeclass and type.

5. In Haskell we have *derived instances* so that obvious or common typeclasses, such as Eq, Enum, Ord, and Show can have the instances generated based only on how the datatype is defined. This is so programmers can make use of these conveniences without writing the code themselves, over and over.

6.16 Typeclass inheritance, partial

This is not a complete chart of typeclass inheritance. It illustrates the relationship between a few of the typeclasses we've talked about 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.
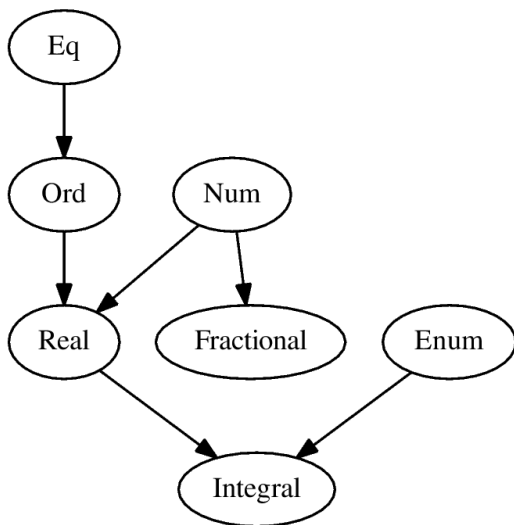
6.17 Follow-up resources

1. P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc.
   `http://www.cse.iitk.ac.in/users/karkare/courses/2010/cs653/Papers/ad-hoc-polymorphism.pdf`

2. Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Typeclasses 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 much more we need to say about them.

A function is an instruction for producing an output from an input, or argument. Functions are *applied* to their arguments, and arguments are passed to functions, and then evaluated to produce the output or result. In this chapter we will demonstrate

- Haskell functions are first-class entities that

- can be values in expressions, lists, or tuples;

- can be passed as parameters to a function;

- can be returned from a function as a result;

- make use of syntactic patterns.

7.2 Arguments

As you know from our discussion of currying, functions in Haskell may appear to take multiple arguments 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 accept an argument and return a result.

All Haskell values can be arguments to functions, including functions. Not all programming languages allow this, but as we will see, it is not as surprising as it may seem. 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.

Note: We use the terms "argument" and "parameter" interchangeably in this book.

Declaring arguments

You declare arguments to functions in Haskell by putting the name of the argument between the name of the function, which is always at the left margin, and the equals sign, separating the name of the argument from both the function name and the equals sign with white space.

First we'll define a value with no arguments:

```haskell
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 just a value without any arguments.

Now let's introduce an argument named `f`:

```haskell
myNum :: Integer
myNum = 1

myVal f = myNum
```

And let's see how that changed the type:

```
Prelude> :t myVal
myVal :: t -> Integer
```

By writing `f` after `myVal` we declare an argument which we refer to as
`f`. This changes our type from `Integer` to `t -> Integer`. The type `t` is
polymorphic because we don't do anything with it – it could be anything.
Here we didn't do anything with `f` so the maximally polymorphic type was
inferred. If we do something with `f` the type will change:

```
Prelude> let myNum = 1 :: Integer
Prelude> let myVal f = f + myNum
Prelude> :t myVal
myVal :: Integer -> Integer
```

Now it knows `f` has to be of type `Integer` because we added it to `myNum`.

We can tell a simple value from a function in part because a simple, lit-
eral value is not applied to any arguments, while functions are necessarily
applied to arguments.

Although Haskell functions only take one argument per function, we can
declare multiple arguments in a term-level function definition:

```
myNum = 1
-- [1]

myVal f = f + myNum
--        [2]

stillAFunction a b c = a ++ b ++ c
--            [ 3 ]
```

1. Declaration of a value of type `Num a => a`. We can tell it's not a
   function because it accepts no explicit arguments between the name
   of the declared value and the `=` and the value `1` is not a function.

2. Here `f` is an argument to the function `myVal`. The function type is
   `Num a => a -> a`. If you assign the type Integer to `myNum`, `myNum` and
   `myVal` change to the types `Integer` and `Integer -> Integer` respec-
   tively.

3. Here `a`, `b`, and `c` are arguments to the function `stillAFunction`. 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 add more arguments:

```
Prelude> let myVal f g = myNum
Prelude> :t myVal
myVal :: t -> t1 -> Integer

Prelude> let 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 gave the type inference nothing to go on with respect to what type 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.

7.3   Binding variables to values

We just looked at how we can declare arguments for defining functions. Now we'll consider how the binding of variables works. We bind a variable to a value when we apply it. When we apply a type variable to an argument, we bind it to a type, whether concrete or constrained-polymorphic. When we apply a function argument to a value, we bind the argument to the value. The binding of variables concerns not only the application of a function argument, 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 `Integer` result until the `addOne` function is applied to an `Integer` value argument. When the argument referred to as `x` is applied to a value, we say that `x` is now *bound* to the value the function was applied to. The function is applied to its argument, but it can't return a result until we bind the variable argument to a value:

```
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 argument 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
```

`y` in `show y` is in scope because the `let` expression binds the variable `y` to `5`. `y` is only in scope *inside* the let expression. Let's kick around some examples to demonstrate 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, "Not in scope: 'y'". 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*:

```haskell
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` was 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 what values referred to by name are and where they come from. Let's annotate the previous example and we'll see what is meant here:

```haskell
bindExp :: Integer -> String
bindExp x = let x = 10; y = 5 in "x: " ++ show x
--      [1]     [2]                              [3]
              ++ " y: " ++ show y
```

1. The argument `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> let x = 5
Prelude> let y = x + 5
Prelude> y
10
Prelude> y * 10
100
Prelude> let 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 which shadowed the *name* y and rebound it. Now, when we enter 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. If no other value is specified as an argument to z, as in the last example, the value of y from the outer scope is passed to z as an argument. But the lexically innermost binding takes precedence, so if a new value is given as an argument to z, that value will take precedence over the outer value (x + 5). (Incidentally, the seeming-sequentiality of defining things in GHCi is, under the hood, actually 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).

7.4   Anonymous functions

We have seen how we give functions a name and call them by name. However, it is also possible to use anonymous functions. In programming, anonymous, or "without a name", refers to the ability to construct objects and use them without naming them.

For example, earlier we looked at this named, i.e., not anonymous, function:

```haskell
triple :: Integer -> Integer
triple x = x * 3
```

And here is the same function but with anonymous function syntax:

```haskell
(\x -> x * 3) :: Integer -> Integer
```

Anonymous functions in Haskell are represented through the use of a special syntax known as lambda syntax represented by the backslash in front of the `x`. Lambda here refers to a lambda abstraction and therefore to the theoretical underpinnings of Haskell and other functional programming languages: the lambda calculus. We used this syntax in the previous chapter when we were talking about manual currying, too.

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:

```haskell
Prelude> let triple = (\x -> x * 3) :: Integer -> Integer
```

Similarly, to apply an anonymous function we'll often need to wrap it in parentheses so that our intent is clear:

```haskell
Prelude> (\x -> x * 3) 1
3
```

```
Prelude> \x -> x * 3 1
```

```
Could not deduce (Num (a0 -> a))
  arising from the ambiguity check for 'it'
from the context (Num (a1 -> a), Num a1, Num a)
  bound by the inferred type for 'it':
    (Num (a1 -> a), Num a1, Num a) => a -> a
  at <interactive>:9:1-13
The type variable 'a0' is ambiguous
When checking that 'it'
  has the inferred type 'forall a a1.
    (Num (a1 -> a), Num a1, Num a) =>
      a -> a'
Probable cause: the inferred type is ambiguous
```

The type error `Could not deduce (Num (a0 -> a))` is because you can't use `Num a => a` values as if they were functions. To the computer, it looks like you're trying to apply `3 1` as if `3` were a function. Here the `it` referred to is `3 1` which it thinks is `3` applied to `1` as if `3` were a function.

Intermission: Exercises

Note 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`

Next, we'll practice writing anonymous lambda syntax.

For example, one could rewrite:

```haskell
addOne x = x + 1
```

Into:

```haskell
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.

```haskell
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:

```haskell
addFive x y = (if x > y then y else x) + 5
```

c) Rewrite the following so that it doesn't use anonymous lambda syntax:

```haskell
mflip f = \x -> \y -> f y x
```

7.5 Pattern matching

Pattern matching is an integral and ubiquitous feature of Haskell. Simply put, it 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 and dispatch on 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 had cause to explore 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`, this 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, just 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
```

```
<interactive>:9:33: Warning:
    Pattern match(es) are overlapped
    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: :50:33-48:
  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| :}

<interactive>:28:5: Warning:
    This binding for 'isItTwo' shadows
    the existing binding
      defined at <interactive>:20:5

<interactive>:28:5: Warning:
    Pattern match(es) are non-exhaustive
    In an equation for 'isItTwo':

    Patterns not matched: #x with #x `notElem` [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 arguments, and pattern matching can let us expose and make use of the data in those arguments.

The next example uses `newtype` which is a special case of `data` declarations. `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
```

Here 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`s, `Username` and `AccountNumber`. We can use pattern matching to break down not only `RegisteredUser`'s contents, but also that of the `newtypes` 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
[1 of 1] Compiling RegisteredUser
Ok, modules loaded: RegisteredUser.
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 perhaps not felici-
tous, as they 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` "callen" and the `Integer` 10456:

```
Prelude> let myUser = (Username "callen")
Prelude> let myAcct = (AccountNumber 10456)
Prelude> printUser $ RegisteredUser myUser myAcct
callen 10456
```

Through the use of pattern matching, we were 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
  | Antarctic
  | 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:

```haskell
-- is it South Africa? If so, return True
isSouthAfrica :: WherePenguinsLive -> Bool
isSouthAfrica SouthAfrica = True
isSouthAfrica Galapagos = False
isSouthAfrica Antarctic = 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:

```haskell
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:

```haskell
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 just return the `WherePenguinsLive` that's inside:

```haskell
humboldt = Peng SouthAmerica
gentoo = Peng Antarctic
macaroni = Peng Antarctic
little = Peng Australia
galapagos = Peng Galapagos
```

Now a more elaborate example. We'll expose the contents of `Peng` and match on what `WherePenguinLives` value we care about in one pattern match:

```haskell
galapagosPenguin :: Penguin -> Bool
galapagosPenguin (Peng Galapagos) = True
galapagosPenguin _                = False


antarcticPenguin :: Penguin -> Bool
antarcticPenguin (Peng Antarctic) = True
antarcticPenguin _                = False

-- in this final function, the || operator
-- is an `or` function, which 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 Basic Datatypes?

```haskell
f :: (a, b) -> (c, d) -> ((b, d), (a, c))
f = undefined
```

When you did that exercise, you may have written it like this:

```haskell
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 clearer and nicer to read version of it:

```haskell
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:

```haskell
-- matchingTuples1.hs
module TupleFunctions where

-- These have to be the same type because
-- (+) is 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, modules loaded: TupleFunctions.
```

Now we're going to use GHCi's `:browse` to see a list of the type signatures and functions we loaded 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, [])
[]
```

Intermission: Exercises

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:

   ```
   -- Remember: Tuples have the same syntax for their
   -- type constructors and their data constructors.

   f :: (a, b, c) -> (d, e, f) -> ((a, d), (c, f))
   f = undefined
   ```

7.6 Case expressions

Case expressions are a way, similar in some respects to `if-then-else`, 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. When we consider the datatype `Bool`:

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

1. Type constructor, we only use this in type signatures, not in ordinary term-level code like case expressions.

2. Data constructor for the inhabitant of `Bool` named `False` – we can match on this.

3. Data constructor for the inhabitant 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 must 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 inputs possible!

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 easily rewrite this as a case expression, matching on the constructors of `Bool`:

```haskell
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 simple case expression to tell us whether or not something is a palindrome:

```haskell
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`:

```haskell
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:

```haskell
-- 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 have been fairly basic and rely on pattern matching with `True` and `False` explicitly. In an upcoming section, we'll look at a more complex use of a case expression.

Intermission: Exercises

We're going to practice using case expressions by rewriting functions. Some of these functions you've already 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 into GHCi to check, rather than trying to do them directly into the REPL.

First, rewrite `if-then-else` expressions into case expressions.

1. The following should return `x` when `x` is greater than `y`.

   ```haskell
   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.

   ```haskell
   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.

1. The following compares a value, x, to zero and returns an indicator for whether x is a postive number or negative number. But 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
```

7.7 Higher-order functions

Higher-order functions (HOFs) are functions that accept functions as arguments. Functions are just 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.

First let's examine a simple higher-order function, flip:

```
Prelude> :t flip
flip :: (a -> b -> c) -> b -> a -> c
Prelude> let sub = (-)
Prelude> sub 10 1
9
Prelude> let rSub = flip sub
Prelude> rSub 10 1
-9
Prelude> rSub 5 10
5
```

The implementation of flip is simple:

```
flip :: (a -> b -> c) -> b -> a -> c
flip f x y = f y x
```

Alternately, it could've been written as:

```
myFlip :: (a -> b -> c) -> b -> a -> c
myFlip f = \ x y -> f y x
```

There's no semantic, or meaning, difference between `flip` and `myFlip`: one declares arguments 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 thinking about 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 merely 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 't0 -> t1 -> t2 -> t2'
            with actual type 'd'

'd' is a rigid type variable bound by
the type signature for
  returnBroke :: (((a -> b) -> c) -> d) -> d

Relevant bindings include
  returnBroke :: (((a -> b) -> c) -> d) -> d

The equation(s) for 'returnBroke' have four arguments,
but its type '(((a -> b) -> c) -> d) -> d' has only one
```

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`.[1]

We *can* have a type that is parenthesized in that fashion as long as we want to do something different than 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 arguments 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`.

---

[1]Fun fact: `returnBroke` is an impossible function.

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` typeclass:

```
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`:

```haskell
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 named `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 just 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:

```haskell
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 catchall), the result will be `GT` or greater than. In a case where `Coder` is the second value passed, this function will return a `LT`, or less than, result. 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> employeeRank codersRuleCEOsDrool Coder CEO
Coder is the boss of CEO
Prelude> employeeRank codersRuleCEOsDrool 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 were able to rely on the behavior of `compare` but make changes in the part we wanted to change. This is the value of HOFs. They give us the beginnings of a powerful method for reusing and composing code.

Intermission: Exercises

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 it in your head, verify in your REPL after you think you have an answer.

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`

7.8   Guards

We have spent a lot of time playing 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 the Basic Datatypes chapter. Note, `if-then-else` is *not* guards! We're just reviewing something similar before moving 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> let x = 0
Prelude> if (x + 1 == 1) then "AWESOME" else "wut"
"AWESOME"
```

The next couple of examples will demonstrate how to write multi-line blocks of code into GHCi – the `:{ :}` notation isn't necessary in source files:

```
-- alternately
Prelude> let x = 0
Prelude> :{
Prelude| if (x + 1 == 1)
Prelude|    then "AWESOME"
Prelude|    else "wut"
Prelude| :}
"AWESOME"
```

The indentation isn't required:

```
Prelude> let x = 0
Prelude> :{
Prelude| if (x + 1 == 1)
Prelude| then "AWESOME"
Prelude| else "wut"
Prelude| :}
"AWESOME"
```

In the exercises at the end of that chapter, 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:

```haskell
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`:

```haskell
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:

```haskell
   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`

2. Introducing an argument for our function and naming it `x`

3. Here's where it gets different. Rather than an `=` immediately after the introduction of any argument(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 to see if this branch should be evaluated or not. The guard case expression between the `|` and `=` must evaluate to `Bool`.

5. The `=` here 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 just another name for `True`, used here as a fallback case in case `x < 0` was `False`.

9. Another `=` to begin declaring the expression to return if we hit the `otherwise` case.

10. We kick `x` back out if it wasn't less than `0`.

Let us demonstrate how this evaluates briefly:

```
*Main> myAbs (-10)
10
*Main> 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. 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:

```haskell
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 takes 3 numbers and tells you if the triangle whose sides they measure is a right triangle or not (using the Pythagorean theorem):

```haskell
-- c is the hypotenuse of the triangle. Google it.

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:

```haskell
dogYrs :: (Num a, Ord a) => a -> a
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 actually equivalent to a 6- or 7-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` expressions with guard blocks. Let's say you gave a test that had 100 questions and you wanted a simple function for translating the number of questions the student got right into a letter grade:

```haskell
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` expression. By defining it there, it is in scope for all the guards above it. There were 100 problems on the hypothetical test, so any x we give it will be divided by 100 to return the letter grade.

Also notice we left out the `otherwise`; we could have used it for the final case but chose instead to use `less than`. That is fine because in our guards we've handled 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 very difficult to reason about it, so it is wise to use `otherwise` in your final guard.

Intermission: Exercises

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 more clear if you rewrite it as an actual `otherwise`

match: `| otherwise = 'F'`. What happens now if you pass a 90 as an argument? 75? 60?

2. What happens if you take `avgGrade` as it is written and reorder the guards? Does it still typecheck and work the same? Try moving `| y >= 0.7 = 'C'` and passing it the argument 90, which should be an 'A.' Does it return an 'A'?

3. The following function returns

```
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. The following function returns

```
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 zero

   d) binary machine language

7. What types of arguments can `numbers` take?

8. What is the type of the function `numbers`?

7.9 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 very concise style, in keeping with the terse functional style Haskell is known for. 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. is a function from 'b' to 'c', passed as an argument (thus the parentheses)

2. is a function from 'a' to 'b'

3. is a value of type 'a', the same as **[2]** expects as an argument

4. is a value of type 'c', the same as **[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. given 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've stitched the result of one function into being the argument of another.

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, `(.)`, 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 the `(.)` or composition operator as being a way of pipelining data through multiple functions. The following composed functions will first add the values in the list together then negate the result of that:

```
Prelude> negate . sum $ [1, 2, 3, 4, 5]
-15

-- which is evaluated like this
negate . sum $ [1, 2, 3, 4, 5]

-- note: this code works as well
negate (sum [1, 2, 3, 4, 5])
negate (15)
-15
```

Notice that we did 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**. Simple!

You may be wondering why we need the $ operator. You might remember way back when we talked about the precedence of various operators that

we said that operator 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 left white space as our function application, this would be evaluated like this:

```
negate . sum [1, 2, 3, 4, 5]
negate . 15
```

In other words, we'd be trying to pass a numeric value where our composition operator needs a function. By using the $ we signal that application to the parameters 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 just a question of style and ease of writing and 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 add the $ operator. It will then look like this:

```
Prelude> take 5 . enumFrom $ 3
[3,4,5,6,7]
```

You may also define it this way, which is more similar to how composition is written in source files:

```
Prelude> let f x = take 5 . enumFrom $ x
Prelude> f 3
[3,4,5,6,7]
```

You may be wondering why 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 parameter of `take`. Feel free to experiment with varying any of the parameters.

```
Prelude> take 5 . filter odd . enumFrom $ 3
[3,5,7,9,11]
```

As you compose more functions, you can 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 "pointfree."

7.10   Pointfree style

Pointfree refers to a style of composing functions without specifying their arguments. The "point" in "pointfree" 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, pointfree 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. Pointfree is just an extension of that idea but now we drop the arguments 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 pointfree style some of the functions we used in the section above:

```
Prelude> let 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 `take 5 . filter odd . enumFrom` $ 3 as a pointfree function?

```
Prelude> let f = take 5 . filter odd . enumFrom
Prelude> f 3
[3,5,7,9,11]
```

And now because we named the function, it can be reused with different numeric arguments.

Here is another example of a short pointfree 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> let f = length . filter (== 'a')
Prelude> f "abracadabra"
5
```

Next, we'll look at a lengthier set of functions, or module, that relies on both composition and pointfree style:

```haskell
-- 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 GHC and see if your results were accurate.

You should now have a good understanding of how you can use `(.)` 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.

7.11 A demonstration of function composition

You may recall that back in the chapter on Printing Strings, 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 parameters, while `print` takes a constrained polymorphic argument, `Show a => a`. The first two work fine, then, if what we need to display on the screen are already strings, but how do we display numbers (or other non-string values)? First we have to convert those numbers to strings, then we can print the strings.

You may also recall a function from our discussion of the `Show` typeclass 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.

Here are two ways to implement `print` with `putStrLn` and `show`:

```
print :: Show a => a -> IO ()
print a = putStrLn (show a)

-- using the . operator for composing functions.
(.) :: (b -> c) -> (a -> b) -> a -> c

-- we can write print as:
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]

-- If we replace the variables with the specific
-- types they take on in this application of (.)

(.) :: Show a => (String -> IO ()) -> (a -> String) -> a -> IO ()
(.) ::          (   b    ->   c) -> (a ->     b) -> a -> c
```

1. is the **String** that **putStrLn** accepts as an argument

2. is the **IO ()** that **putStrLn** returns, that is, performing the side effect of printing and returning unit.

3. is `a` who must implement the `Show` typeclass, this is the `Show a => a` from the `show` function which is a method on the `Show` typeclass.

4. is the `String` which `Show` returns. This is what the `Show a => a` value got stringified into.

5. is the `Show a => a` the final composed function expects.

6. is the `IO ()` the final composed function returns.

We can now make it pointfree. When we are working with functions primarily in terms of *composition* rather than *application*, the pointfree version can often, but not always, be more elegant.

Here's the pointfree version of `print`:

```
-- Previous version of the function
print :: Show a => a -> IO ()
print a = (putStrLn . show) a

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 pointfree function. Saying that we could apply `putStrLn . show` to an argument in this case is redundant.

7.12 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 arguments 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 function of the type `a -> a -> Bool -> a` once each 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 y True  = x
foldBool3 x y False = y
```

3. Fill in the definition. Note that the first argument to our function is *also* a function which 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 pointfree versions of existing code. This involves some new information, so read the following explanation carefully.

   Typeclasses are dispatched by type. Read is a typeclass like Show, but it is the dual or "opposite" of Show. In general, the Read typeclass isn't something you should plan to use a lot, but this exercise is structured to teach you something about the interaction between typeclasses and types.

   The function read in the Read typeclass has the type:

   ```
   read :: Read a => String -> a
   ```

   Notice a pattern?

   ```
   read :: Read a => String -> a
   show :: Show a => a -> String
   ```

   Write 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

   -- id :: a -> a
   -- id x = x

   roundTrip :: (Show a, Read a) => a -> a
   roundTrip a = read (show a)

   main = do
     print (roundTrip 4)
     print (id 4)
   ```

5. Next, write a pointfree version of roundTrip.

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 `Int`, so GHC will use the `Int` instance of the `Show` typeclass to stringify our `Int` of `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 now? Make the application of your pointfree version of `roundTrip` to the argument `4` on line 10 work. You will only need the *has the type* syntax of `::` and parentheses for scoping.

7.13 Chapter Definitions

1. *Binding* or *bound* is a common word used to indicate connection, linkage, or association between two objects. In Haskell we'll use it to talk about what value a variable has. Bindings as a plurality will usually refer to a collection of variables and functions which can be referenced by name.

   ```
   blah :: Int
   blah = 10
   ```

   Here the variable `blah` is bound to the value `10`.

2. An *anonymous function* is a function which 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 which each take one argument and one result. This is accomplished through the 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

-- 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 syntax handling products and sums. 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.

```
-- nullary data constructor, not a sum or product.
-- Just a single value.
data Blah = Blah
```

Pattern matching on Blah can only do one thing.

```haskell
blahFunc :: Blah -> Bool
blahFunc Blah = True

data Identity a =
  Identity a
  deriving (Eq, Show)
```

Identity is a unary data constructor. Still not a product, only contains one value.

```haskell
-- when you pattern match on Identity
-- you can unpack and expose the 'a'

unpackIdentity :: Identity a -> a
unpackIdentity (Identity x) = x


-- But you can choose to ignore
-- the contents of Identity

ignoreIdentity :: Identity a -> Bool
ignoreIdentity (Identity _) = True


-- or ignore it completely since matching on
-- a non-sum data constructor changes nothing.

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*:

```haskell
productUnpackOnlyA :: Product a b -> a
productUnpackOnlyA (Product x _) = x


productUnpackOnlyB :: Product a b -> b
productUnpackOnlyB (Product _ y) = y
```

Or can we can bind them both to a different name:

```haskell
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?

```haskell
data SumOfThree a b c =
    FirstPossible  a
  | SecondPossible b
  | ThirdPossible  c
  deriving (Eq, Show)
```

Now we can discriminate by the inhabitants of the sum and choose to do different things based on which constructor in the sum they were.

```haskell
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:

```haskell
-- 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"

-- 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.

```
-- Technically higher order
-- because of currying
Int -> Int -> Int

-- See? 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]

-- also higher order, this one
-- takes a function argument which itself
-- is higher order as well.

((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:

```haskell
(.) :: (b -> c) -> (a -> b) -> a -> c

-- is actually

(.) :: (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`, `f` is applied to the result of `g x`.

8. *Pointfree* 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 pointfree is an argument.

```haskell
-- not pointfree
blah x = x
addAndDrop x y = x + 1
reverseMkTuple a b = (b, a)
reverseTuple (a, b) = (b, a)
wtf d = zipWith (+) (\ l -> (map d l) >>= \ h -> h)

-- pointfree versions of the above
blah = id
addAndDrop = const . (1 +)
reverseMkTuple = flip (,)
reverseTuple = uncurry (flip (,))
wtf = zipWith (+) . (join .) . map
```

To see more examples like this, check out the Haskell Wiki page on Pointfree at `https://wiki.haskell.org/Pointfree`.

7.14 Follow-up resources

1. Paul Hudak; John Peterson; Joseph Fasel. A Gentle Introduction to Haskell, chapter on case expressions and pattern matching.
   `https://www.haskell.org/tutorial/patterns.html`

2. Simon Peyton Jones. The Implementation of Functional Programming Languages, pages 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, page 11 for explanation of currying.
   `http://www.cs.cmu.edu/~crary/819-f09/Strachey67.pdf`

4. J.N. Oliveira. An introduction to pointfree 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

Functions that call themselves

> 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 allowing 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 which 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 ability 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 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've seen expressions like `4!`. The *bang* you're seeing next to the number 4 is the notation for the factorial function.

Lets evaluate `4!`:

```
4! = 4 * 3 * 2 * 1
        12 * 2 * 1
            24 * 1
                24
4! = 24
```

Now let us do this the silly way in Haskell:

```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.

Now we'll 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)

-- To make my point obvious, 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 which 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
```

Because we have the base case `factorial 0 = 1` in the fixed version, here is how our 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
```

Some hand-waving is needed in Haskell due to `bottom`, but proper code intended for production use shouldn't have intentional bottom values anywhere.

Another way to look at recursion

In the last chapter, we looked at a higher-order function called composition. 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 the 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 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 the compose function `(.)` is written:

```
(.)    :: (b -> c) -> (a -> b) -> a -> c
(.) f g = \x -> f (g x)
```

---

[1]Many thanks to George Makrydakis for discussing this with us.

Any programming language, such as Haskell, that is built purely on lambda calculus has only one verb: *apply* a function to an argument. Applying a function to an argument and returning a result is all we can ever really do, no matter what syntactic conveniences we construct to make it seem that 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 function application in the same way that composition is. 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.[2]

Let's look at some code to see the similarity in patterns:

```
inc :: Num a => a -> a
inc = (+1)

three = inc . inc . inc $ 0
-- different syntax, same thing
three' = (inc . inc . inc) 0
```

---

[2]Because Haskell is built on pure lambda calculus, recursion is implemented in the language through the Y, or fixed-point combinator. You can read a very good explanation of that at `http://mvanier.livejournal.com/2897.html` if you are interested in knowing how it works.

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:

```haskell
incTimes :: (Eq a, Num a) => a -> a -> a
incTimes 0 n = n
incTimes times n = 1 + (incTimes (times - 1) n)
```

Here, `times` is 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 zero times, it will just return our `n` 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 this, 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 zero, it returns `n` and that's all the applications we get.

We can abstract the recursion out of `incTimes`, too:

```haskell
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`:

```haskell
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 didn't use `(.)`.

```haskell
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 failed with an error or those that failed to terminate. In logic, ⊥ corresponds to "false". Let us examine a few ways by which we can have bottom in our programs:

```
Prelude> let x = x in x
*** Exception: <<loop>>
Prelude>
```

Here GHCi detected that `let x = x in x` was never going to return and short-circuited the never-ending computation. This is an example of bottom because it was never going to return a result. Note that if you're using a Windows computer, this example may just freeze your GHCi and not throw an exception.

Next let's define a function that will return an exception:

```haskell
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 evaluated `f False` and got `0`, that didn't result in a "bottom" value. But, when we evaluated `f True`, we got an exception which is a means of expressing that a computation failed. We got an exception because we specified 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:

```haskell
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> let f :: Bool -> Int; f False = 0
Prelude> f False
0
Prelude> f True
*** Exception: 6:23-33:
        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 which 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 which 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
```

We see that **Maybe** can take an argument. 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're wanting. **Maybe** makes all uses of **nil** 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:

```haskell
f :: Bool -> Maybe Int
f False = 0
f _ = Nothing
```

```
Prelude> :l code/brokenMaybe1.hs
[1 of 1] Compiling Main

code/brokenMaybe1.hs:3:11:
    No instance for (Num (Maybe Int)) arising from the literal '0'
    In the expression: 0
    In an equation for 'f': f False = 0
```

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:

```haskell
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

code/brokenMaybe2.hs:3:11:
    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
```

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 calculates the nth number in a Fibonacci sequence. The Fibonacci sequence, as you may know, is a sequence of numbers in which each number is the sum of the previous two: `1, 1, 2, 3, 5, 8, 13, 21, 34...`. Precisely because we have an indefinite computation that relies on adding two of its own members, 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.

1. 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 wanting to return the value that is 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 typeclass for constrained polymorphism. Specifically, we would want a type signature that takes one integral argument and returns one integral result. So, our type signature will look something like this:

   ```haskell
   fibonacci :: Integer -> Integer
   -- or
   fibonacci :: Integral a => a -> a
   ```

2. 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 zero. When the recursive process hits zero, it should terminate.

The Fibonacci sequence is actually a bit trickier than some, though, because it really 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 zero is 1, and we add zero to 1 to start the sequence so those will be our base cases:

```
fibonacci :: Integral a => a -> a
fibonacci 0 = 0
fibonacci 1 = 1
```

3. 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 an argument to this function then.

   But that argument will also be used as arguments 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 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 quite yet.
```

4. 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.

```haskell
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?

```
*Main> 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 really 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 call the function to itself twice.

```haskell
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:

```
*Main> fibonacci 6
8
```

Why? Because it evaluates this 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
```

Zero and 1 have been defined as being equal to zero and 1. So here our recursion stops, and it starts adding 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 the function will make reference to its own results in a repeated fashion.

8.5 Writing our own integral division function

Many people learned multiplication by memorizing multiplication tables, usually up to 10x10 or 12x12 (dozen). 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 evaluate `10 / 5` to get the answer `2`, `10` is the numerator, `5` the denominator, and `2` was 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 (that won't change your code) to make it more obvious if you wished:

```
dividedBy :: Integer -> Integer -> Integer
dividedBy = div

-- changes to

type Numerator = Integer
type Denominator = Integer
type Quotient = Integer

dividedBy :: Numerator -> Denominator -> Quotient
dividedBy = div
```

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 minute. 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 subtracted repeatedly 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 stopped.
-- We subtracted 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 subtracted 4 six times and had 1 as our remainder. We can generalize this process of dividing whole numbers, returning the quotient and remainder, into a recursive function which does the repeated subtraction and counting for us. Since we'd like to return the quotient *and* the remainder, we're going to return the 2-tuple **(,)** 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)
```

First, take a look at the type signature. We've changed it from the one we had originally worked out, both to make it more polymorphic (Integral a => a versus Integer) and also to return the tuple instead of just an integer.

Here we used 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 zero 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 subtracted.
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 was 0, there's no remainder. 10 divides into 2 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)
  -- otherwise is literally the value True
  -- so if first branch fails, this always succeeds

    go 8 2 1
      -- 8 isn't < 2, so 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, so the otherwise branch
       -- n == 4, d == 2, count == 3

        go 4 2 3
          go (4 - 2) 2 (3 + 1)
         -- 4 isn't < 2, so the otherwise branch
         -- n == 2, d == 2, count == 4

          go 2 2 4
            go (2 - 2) 2 (4 + 1)
           -- 2 isn't < 2, so 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 was 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 following function

   ```
   func    :: [a] -> [a] -> [a]
   func x y = x ++ y
   ```

   which of the following is true?

   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 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 that if n was 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 was 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 really we ought to use a datatype that lets us say there was no sensible result when the user divides by zero. If you need inspiration, consider using the following datatype to handle this.

```haskell
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.

$$MC(n) = \begin{cases} n - 10 & \text{if } n > 100 \\ MC(MC(n + 11)) & \text{if } n \leq 100 \end{cases}$$

```haskell
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. It'll be explained 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

```haskell
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 which 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 1-9 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 deeper 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 output should look in the REPL when it's working:

```
Prelude> wordNumber 12324546
"one-two-three-two-four-five-four-six"
Prelude>
```

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.

```haskell
-- not recursive
lessOne :: Int -> Int
lessOne n = n - 1

-- recursive
zero :: Int -> Int
zero 0 = 0
zero n = zero (n - 1)
```

Chapter 9

Lists

This thing and some more stuff

> 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 list's datatype and how to pattern match on lists;

- practice many standard library functions for operating on lists;

- learn about the underlying representations 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*). Here cons 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 `(:)` `` `cons` `` is a *product* because it takes two arguments. Briefly, a sum type can be read as an "or" as in the Bool datatype where you get 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 the value in an additive fashion, 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. takes a single type constructor argument 'a'

3. at the term level can be constructed via

4. nullary constructor `[]`

5. *or* it can be constructed by

6. data constructor `(:)` which is a product of a value of the type `a` we mentioned in 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 a programmer, 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> let myHead (x : _) = x
Prelude> :t myHead
myHead :: [t] -> t
Prelude> myHead [1, 2, 3]
1
```

We can do the opposite as well:

```
Prelude> let 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. Both `myHead` and `myTail` have refutable patterns and no fallback – 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:

```haskell
myTail         :: [a] -> [a]
myTail []      = []
myTail (_ : xs) = xs
```

In that case, our function now evaluates like this (the prompt isn't Prelude because we loaded this from a source file):

```
*Main> myTail [1..5]
[2,3,4,5]
*Main> 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:

```haskell
safeTail        :: [a] -> Maybe [a]
safeTail []     = Nothing
safeTail (x:[]) = 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` to reflect the fact that

if your list has only one value inside it, it doesn't have a tail – it only has a head. If you leave this case out, then this function will return `Just []` for lists that have only a head. Take a few minutes to play around with this 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 building lists in terms of the successive applications of 'cons' (:) 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. Ranges come with — surprise, surprise — some associated syntactic sugar. Here are some examples using the syntactic sugar, followed by the desugared equivalents using the `enumFromTo` function:

```
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 typeclass. One thing to note is that the first two functions, `enumFrom` and `enumFromThen`, generate lists of unspecific, 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.

Be aware that `enumFromTo` must have its first argument be "lower" than the second argument.

```
Prelude> enumFromTo 3 1
[]
Prelude> enumFromTo 1 3
[1,2,3]
```

Otherwise you'll just get an empty list.

Exercise

Some things you'll want to know about the Enum typeclass:

```
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`. Do not use range syntax to do so. It should return the same results as if you did `[start..stop]`. In the interest of keeping it sporting, we're going to provide the type, add a typeclass constraint on *a* to make it a little easier, and suggest that you may want to make use of a sub-function defined within `myEnumFromTo`. If you find a different way, that's okay too! You'll want to use `succ` from Enum as well.

```
myEnumFromTo :: Enum a => a -> a -> [a]
myEnumFromTo = 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 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 just 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 are, such as Integer, an infinite set. But as long as we're only taking a certain number of elements from that, 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 takes an Int argument, applies it to a list, cuts the 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 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.

```
Prelude> takeWhile (<3) [1..10]
```

```
[1,2]
-- Takes the elements that are less than 3

Prelude> takeWhile (<8) (enumFromTo 5 15)
[5,6,7]
-- Takes the elements that are less than 8

Prelude> takeWhile (>6) [1..10]
[]
-- It 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 (=='a') "abracadabra"
"a"
```

In the final example above, why does it only return a single a?

Finally, we'll look at `dropWhile` whose 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"
```

Intermission: Exercises

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:

   ```
   *Main> myWords "all i wanna do is have some fun"
   ["all","i","wanna","do","is","have","some","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):

```haskell
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

-- 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

-- This is what 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 it.

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 functions 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 what 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 met the True case in the predicate.

For example, let's say we wanted a similar list comprehension as we used above, but this time we wanted 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 2, 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 wanted 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> [x^y | x <- [1..10], y <- [2, 3], x^y < 200]
[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 the same length or, due to the nature of the tuple type, even the same type:

```
Prelude> [(x, y) | x <- [1, 2, 3], y <- [6, 7]]
[(1,6),(1,7),(2,6),(2,7),(3,6),(3,7)]

Prelude> [(x, y) | x <- [1, 2, 3], y <- ['a', 'b']]
[(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 generated a list of all the values of x^2 when x is a number from 1-10. Let's say you wanted to use that list in another list comprehension. First, you'd want to give that list a name. Let's call it "mySqr":

```
Prelude> let 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 10 for the sake of brevity:

```
Prelude> let mySqr = [x^2 | x <- [1..10]]
Prelude> [(x, y) | x <- mySqr, y <- [1..3], x < 10]
[(1,1),(1,2),(1,3),(4,1),(4,2),(4,3),(9,1),(9,2),(9,3)]
```

Intermission: Exercises

Take a look at the following functions, figure 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` 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> [x | x <- "Three Letter Acronym", elem x ['A'..'Z']]
"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:

```
Prelude> let f xs = [x | x <- xs, elem x ['A'..'Z']]
```

We name the function f just to keep things short and simple, and 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 necessary.

All right, so we have our f function with which we can generate acronyms from any string:

```
Prelude> f "Self Contained Underwater Breathing Apparatus"
"SCUBA"
Prelude> f "National Aeronautics and Space Adminstration"
"NASA"
```

Given the above, what do you think this function would do:

```
Prelude> let myString xs = [x | x <- xs, elem x "aeiou"]
```

Intermission: Exercises

Given the following:

```
Prelude> let mySqr = [x^2 | x <- [1..5]]
Prelude> let 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 function so that it only uses the x and y values that are less than 50.

3. Now 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 the recursive cons `(:)` operators. Given the data: `[1, 2, 3]`, we get a list that looks like:

```
1 : 2 : 3 : []
or
1 : (2 : (3 : []))
```

```
   :
  / \
 1   :
    / \
   2   :
      / \
     3  []
```

The problem with the `1 : (2 : (3 : []))` 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 just 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. Constructing the list when that is necessary, however, 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 3 and empty list. But when we need to build the list, to print it in the REPL for example, it proceeds from the bottom of the list up the spine, first putting the 3 into the empty list, then adding the 2 to the front of that list, then, finally, putting the 1 in the front of that. Because Haskell's evaluation is nonstrict, the list isn't constructed until it's consumed – indeed, nothing is evaluated until it must be. Until it's consumed or you force strictness in some way, there are 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 that aren't just lists. In the case of 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 2 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 which introduce strictness to make code faster when it won't change how your code evaluates. Additionally polymorphism means values like `Num a => a` are really waiting for a sort of "argument" which will make it concrete. To avoid this, you have to assign a more concrete type such as Int or Double, otherwise it'll stay 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> let 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 just look at a few examples to get a sense for what might be going on.

Below we list some expressions and whether they are in WHNF, NF, both, or neither:

```
(1, 2) -- WHNF & NF

(1, _ + _)
-- WHNF, but not NF. The (+) and its
-- unknown arguments could be evaluated

(1, 1 + 1)
-- WHNF, but not NF.
-- The 1 + 1 could be evaluated.

\x -> x * 10 -- WHNF & NF
-- It's in normal form because while
-- (*) 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" -- Neither WHNF nor NF
```

When we define a list and define all its values, it is in NF and all its values are known. There's really nothing left to evaluate:

```
Prelude> let 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" which might either be 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 values does not force evaluation of the rest of the list:

```
Prelude> let 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 used `sprint` we only saw `1 : 2 : _`. Evaluating to normal form would've meant recursing through the entire list forcing not only the entire spine but also the values each cons cell contained.

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 the list even if they don't force evaluation of each value. Pattern matching is strict by default, so pattern matching forces spine strictness. It can evaluate the spine only or the spine as well as the values that inhabit each cons cell, depending on context.

On the other hand, `length` is strict in the spine but not the values. If we defined a list such as `[1, 2, 3]`, 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> let x = [1, undefined, 3]
Prelude> length x
3
```

The first and third values in the list were numbers, but the second value was `undefined` and `length` didn't make it crash. Why? Because `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-matched on the `(:)` data constructor, but wanted to ignore the value which 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 "_", it's 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 it, it'll think you're trying to refer a hole if you use _ on the right-hand side in the definition.

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> let 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 `1` value at the beginning, and attempting to get the length also fails because it can't count undefined spine values.

It's possible to write functions which 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 values in the 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 intuition for Haskell's evaluation strategies takes time and prac-
tice. 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.

Intermission: Exercises

Will it blow up?

1. Will the following expression return a value or be ⊥?

   ```
   [x^y | x <- [1..5], y <- [2, undefined]]
   ```

2. `take` 1 `$` [x^y | x `<-` [1..5], y `<-` [2, undefined]]

3. Will the following expression return a value?

   ```
   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; or,

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 the 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 typeclass named `Functor` and can be applied to data other than lists. We will learn more about `Functor` later; for now, we'll focus just 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)
-- (a -> b) becomes more specific, is resolved 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 this has the Functor typeclass constraint

fmap (+1)
-- again, (a -> b) is now more specific

Prelude> :t fmap (+1)
fmap (+1) :: (Num b, Functor f) => f b -> f b
-- a bit different from map because the Functor
-- typeclass includes more than just lists!
```

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 `[]` empty list case because List 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 typechecker 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 nonspecific 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 `(:)` 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 actually applies the function argument to the contents of the list.

7. We `(:)` cons the value returned by the expression `f x` onto the head of the result of `map`'ing 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, which will be covered later, but does give an idea of what's going on:

```
map (+1) [1, 2, 3]
-- desugared, (:) is infixr 5, so it's right-associative
map (+1) (1 : (2 : (3 : [])))

-- Not an empty list, so second pattern-match in map fires.
-- Apply (+1) to value, then map
(+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 : [])))

-- Last time we'll trigger the second-case of map
(+1) 1 : ((+1) 2 : ((+1) 3 : (map (+1) [])))

-- Now we trigger the base-case that handles empty list
-- and return the empty list.
(+1) 1 : ((+1) 2 : ((+1) 3 : []))

-- Now we reduce
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   []
```

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 actually 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 was forced because
-- GHCi printed the list that resulted

Prelude> (+1) undefined
*** Exception: Prelude.undefined

Prelude> (1, undefined)
(1,*** Exception: Prelude.undefined
Prelude> fst $ (1, undefined)
1
```

```
Prelude> map (+1) [1, 2, undefined]
[2,3,*** Exception: Prelude.undefined

Prelude> take 2 $ map (+1) [1, 2, undefined]
[2,3]
```

In the final example, the **undefined** value was never forced and there was no error because we used **take 2** to request only the first two elements. With **map (+1)** we only force as many values as cons cells we forced. 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 later when we talk about non-strictness and data structures. Many Haskell users will never need to worry about this more than a handful of times.

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]
```

Or in this example we map the **take** function over a list of lists to return a list of lists:

```
Prelude> map (take 3) [[1..5], [1..5], [1..5]]
[[1,2,3],[1,2,3],[1,2,3]]
```

Indeed, you can **map** and **fmap** a variety of functions over list structures. Here, we'll map an **if-then-else** 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 Folds chapter.

Intermission: Exercises

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. ```
   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 were 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` that we used before
   ```

c) `map` sum [[1..5], [1..5], [1..5]]

6. Back in the Functions chapter, 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 Prelude prompt.

9.10 Filtering lists of values

When we talked about function composition in the Functions chapter, 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 just 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 met the condition. It's important to remind ourselves that this function, as we can see in the defintion, 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> filter (\x -> (rem x 2) == 0) [1..20]
[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 you try writing some filter functions of your own to get comfortable with the pattern.

Intermission: Exercises

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 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 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 are 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]
```

Just 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) -> [a] -> [b] -> [c]
--              [1]         [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 from 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> zipWith max [10, 5, 34, 9] [6, 8, 12, 7]
[10,8,34,9]
```

Zipping exercises

1. Write your own version of `zip :: [a] -> [b] -> [(a, b)]` and ensure it behaves the same as the original.

2. Do what you did for `zip`, but now for `zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]`

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 pointfree.

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 3 means that 'A' will become 'D' and 'B' will become 'E,' for example. If you did a leftward shift of 5, 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 maxmize 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:

```haskell
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 3 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`[1] function can take 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:

---

[1]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 typeclass Foldable. It still works with lists just the same as it did before. Proceed assured that we'll cover this later.

```haskell
-- 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.

   ```haskell
   myOr :: [Bool] -> Bool
   myOr = undefined
   ```

2. `myAny` returns True if `a -> Bool` applied to any of the values in the list returns True.

   ```haskell
   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`.

   ```haskell
   -- 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 list.

   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 -> "WOOO " ++ [x] ++ " HOOOO ") "blah"
"WOOO b HOOOO WOOO l HOOOO WOOO a HOOOO WOOO h HOOOO "
```

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 returned GT for.

```
-- If you import maximumBy from Data.List, you'll see the type is
-- Foldable t => (a -> a -> Ordering) -> t a -> a
-- rather than
-- (a -> a -> Ordering) -> [a] -> a
-- if you have GHC 7.10 or newer. Seeing a pattern?

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
```

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 returned LT for.

```
-- blah blah GHC 7.10 different type that uses Foldable.

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
```

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.

```haskell
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* 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 consing.

   b) A list of the number 2 followed by the number 3.

   c) The final result of consing 1 onto `[2, 3]`.

   d) The type variable *a* corresponds to `1`, the value we consed 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 `(:)` accepts, which was `[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. *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 nesting of multiple cons cells, possibly indefinitely with the use of recursive functions, for representing an indefinite number of values in series:

```
data [] a = []  |   a : [a]
--                        ^ cons operator

-- Defining it ourselves

data List a = Nil | Cons a (List a)

-- Creating a list using our 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 collection that *isn't* the values contained therein. Often spine will be used in reference to lists, but it applies with tree data structures 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 Answers

Please remember that some of these do not have only one possible answer. If your functions get the same result as ours does when given the same input, you're golden.

Intermission Exercises

1. Using `takeWhile` and `dropWhile`, write a function that takes a string and returns a list of strings. Here is what we were looking for, although the standard library function `words` can be implemented differently.

```
myWords   :: String -> [String]
myWords [] = []
myWords (' ':xs) = myWords xs
myWords xs =
  takeWhile (/= ' ') xs:myWords (dropWhile (/= ' ') xs)
```

2. The next two exercises here are very similar to the above, so use that answer as a guide to the general process and tinker with it until they compile and work the way they're meant to.

Take a look at the following functions, figure what you think the output lists will be, and then run them in your REPL to verify. Your answers here should come from GHCi.

Given the following:

```
Prelude> let mySqr = [x^2 | x <- [1..5]]
Prelude> let myCube = [y^3 | y <- [1..5]]
```

a) First write a function that will make tuples of the outputs of `mySqr` and `myCube`.

   `tupleThem` = [(x, y) | x <- mySqr, y <- myCube]

b) Now alter that function so that it only uses the x and y values that are less than 50.

   `tupleThem` =
       [(x, y) | x <- mySqr, y <- myCube, x < 50, y < 50]

c) Now apply another function to that list comprehension to determine how many tuples inhabit your output list.

   `longTuple` = length tupleThem

3. Will it blow up? For all of these, you should be trying to answer before running them through GHCi and then using your REPL to make sure you were right.

4. Is it in normal form?

   For each expression below, determine whether it's in:

   a) normal form, which implies weak head normal form;

   b) weak head normal form only; or,

   c) neither.

   a) NF, due to being values being fully evaluated.

   b) WHNF, due to _ not being evaluated and the outermost term is (:)

   c) Neither, since enumFromTo hasn't been applied, isn't a data constructor.

    d) Neither, length is the outermost term and isn't a data constructor.

    e) Neither, sum is the outermost part and not a data constructor.

    f) Neither, since ++ is the outermost part and is a function, not a data constructor.

    g) WHNF, outermost part is (,) but the first value is unevaluated.

5. The next section of exercises are mostly ones you need to think about and then check in GHCi, but there is the little matter of rewriting a function using **bool**. We expected something like this:

```
-- the function we're trying to rewrite
map (\x -> if x == 3 then (-x) else (x)) [1..10]

-- with bool
map (\x -> bool x (-x) (x == 3)) [1..10]
```

9.15 Follow-up resources

1. `Data.List` documentation for the **base** library.
   `http://hackage.haskell.org/package/base/docs/Data-List.html`

2. Ninety-nine Haskell problems.
   `https://wiki.haskell.org/H-99:_Ninety-Nine_Haskell_Problems`

Chapter 10

Folding lists

Data structure origami

> 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;

- go into detail the evaluation processes of folds;

- walk through the process of 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` may be different if you have GHC 7.10 or newer:

```
-- 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
```

[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.

Then lined up next to each other:

```
foldr ::                (a -> b -> b) -> b -> [] a -> b
foldr :: Foldable t => (a -> b -> b) -> b -> t  a -> b
```

For now, all you need to know is that GHC 7.10 abstracted out the list-specific part of folding into a typeclass 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> :t foldr
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b

Prelude> let listFoldr = foldr :: (a -> b -> b) -> b -> [] a -> b

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 worked?
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 similiarity? 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 * 1 = 2`. With list concatenation, 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 acc []     = acc
foldr f acc (x:xs) = f x (foldr f acc xs)
```

The similarities between this and the recursive patterns we saw above should be clear. The "rest of the fold," `(foldr f acc xs)` is an argument to the function `f` we're folding with. The `acc` is the accumulator, sometimes called "zero," of our fold. It provides a fallback value for the empty list case and a second argument to begin our fold with. The accumulator 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 acc xs =
  case xs of
    []     -> acc
    (x:xs) -> f x (foldr f acc xs)
```

Here we see how the right fold associates to the right. This will reduce just like the `sum` example from earlier:

```
-- we're reducing:
foldr (+) 0 [1, 2, 3]

-- First step, what's ``xs'' in our case expression?
foldr (+) 0 [1, 2, 3] =
  case [1, 2, 3] of
    ...

-- What case of the expression matches?
foldr (+) 0 [1, 2, 3] =
  case [1, 2, 3] of
    []     -> 0
    (x:xs) -> f x (foldr f acc xs) --<--- this one

-- What are f, x, xs, and acc 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 which doesn't continually force the rest of the fold. If it were to stop on the first case here, then it would've 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 the (foldr (+) 0 [2, 3]):

```
-- there is (+) 1 implicitly wrapped around this
-- continuation of the recursive fold

foldr (+) 0 [2, 3] =
  case [2, 3] of
    []         -> 0 -- this didn't match again
    (2 : [3]) -> (+) 2 (foldr (+) 0 [3])
```

(+) 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:"

```
-- there is (+) 1 ((+) 2 ...) implicitly wrapped
-- around this continuation of the recursive fold

-- Next recursion.
foldr (+) 0 [3] =
  case [3] of
    []        -> 0 -- this didn't match again
    (3 : []) -> (+) 3 (foldr (+) 0 [])
```

We're going to ask for more foldr one last time and then we'll hit our base case:

```
-- there is (+) 1 ((+) 2 ((+) 3 ...)) implicitly wrapped
-- around this continuation of the recursive fold

-- Last recursion, this is the end of the spine.
foldr (+) 0 [] =
  case [] of
    []        -> 0    --<-- This one finally matches.
    -- ignore the other case, didn'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))
```

Just like 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]

```
Prelude> let xs = map show [1..5]
Prelude> foldr (\x y -> concat ["(",x,"+",y,")"]) "0" xs
"(1+(2+(3+(4+(5+0)))))"
```

---

[2] Idea borrowed from Cale Gibbard from the haskell Freenode IRC channel and on the Haskell.org wiki `https://wiki.haskell.org/Fold#Examples`

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, 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 acc (x:xs) = f x (foldr f acc xs)
--                       ^--------------^
--                       rest of the fold
```

Given this two-stage process and non-strict evaluation, if `f` doesn't evaluate its second argument (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 just 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:

```
-- (+) will unconditionally evaluate the entire
-- spine and all of the values
Prelude> foldr (+) 0 [1..5]
15

-- here, we give an undefined value
Prelude> foldr (+) 0 [1, 2, 3, 4, undefined]
*** Exception: Prelude.undefined
Prelude> foldr (+) 0 (take 4 [1, 2, 3, 4, undefined])
10

-- here, the undefined is part of the spine
Prelude> foldr (+) 0 ([1, 2, 3, 4] ++ undefined)
*** Exception: Prelude.undefined
Prelude> foldr (+) 0 (take 4 ([1, 2, 3, 4] ++ undefined))
```

```
10
```

By taking only the first four elements, we stop the recursive folding process
at just the first four values 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 uncondition-
ally, 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> length (take 4 ([1, 2, 3, 4] ++ undefined))
4
```

take 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
it *stops* returning elements of the list it was given when it hits the length
limit you gave it. Consider this:

```
Prelude> length $ take 2 $ take 4 ([1, 2] ++ undefined)
2
```

It doesn't *matter* that **take 4** could've 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** works,
let's consider the first argument:

```haskell
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f acc []     = acc
foldr f acc (x:xs) = f x (foldr f acc xs)
--                          [1]
```

The first argument [1] 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 value 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 its arguments and just 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:

```haskell
Prelude> foldr (\_ _ -> 9001) 0 [1..5]
9001
Prelude> foldr (\_ _ -> 9001) 0 [1, 2, 3, 4, undefined]
9001
Prelude> foldr (\_ _ -> 9001) 0 ([1, 2, 3, 4] ++ undefined)
9001
```

Everything is fine unless the first piece of the spine is bottom:

```haskell
Prelude> foldr (\_ _ -> 9001) 0 undefined
*** Exception: Prelude.undefined

Prelude> foldr (\_ _ -> 9001) 0 [1, undefined]
9001
Prelude> foldr (\_ _ -> 9001) 0 [undefined, undefined]
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 it affects the way our programs evaluate.

Traversing the rest of the spine doesn't occur unless the function asks for the results of having folded the rest of the list. In the following examples, we don't force traversal of the spine because `const` just throws away its second argument, which is the rest of the fold:

```
-- 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 the beginning to the 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 reasons as with `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 so
       ((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> let f = (\x y -> concat ["(",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` value to the head of the list, whereas `foldr` had added 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 the scans and folds are 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.

Effects of left associativity

Next we'll take a closer look at some of the effects of the associativity of `foldl`. In the next set of comparisons, we will use `flip` to avoid a type error, but the reasons may not seem clear. Let's compare these type signatures:

```
foldr :: (a -> b -> b) -> b -> [a] -> b

foldl :: (b -> a -> b) -> b -> [a] -> b

(:) :: a -> [a] -> [a]
```

The type of `(:)` requires that a value be the first argument and a list be the second argument, so the value is then prepended, or "consed onto," the front of that list. When we use `foldr` for the identity of a list, like this:

```
Prelude> foldr (:) [] [1..5]
[1,2,3,4,5]
```

After it traverses the spine and starts reconstructing the list, it looks like this:

```
1 : 2 : 3 : 4 : (5 :  [])
                a -> [a]
```

So the consing process for **foldr** matches the type signature for **(:)**. But the process for **foldl** does not. 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:

```
([]   : 5) : 4 : 3 : 2 : 1
 [a] -> a
```

In other words, the list argument is before the value argument, which is opposite of what **(:)** expects. Using **flip** allows us to flip the order of the arguments so that **foldl** can reconstruct its list:

```
Prelude> foldl (flip (:)) [] [1..5]
[5,4,3,2,1]
```

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 (const 2 (const 3 (const 4 (const 5 0)))))
1
```

The starting place for the folding function is `const 5 0`, and 5 will be the resulting value. The next step is `const 4 5` and 4 will be the result. That process continues until it reaches the beginning of the list and evaluates `const 1 2` and returns the final value of 1.

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
```

Intermission: Exercises

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 them and test 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 calls itself 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> foldr const 0 ([1..5] ++ undefined)
1
Prelude> foldr (flip const) 0 ([1..5] ++ undefined)
*** Exception: Prelude.undefined

Prelude> foldl const 0 ([1..5] ++ undefined)
*** Exception: Prelude.undefined
Prelude> foldl (flip const) 0 ([1..5] ++ undefined)
*** Exception: Prelude.undefined
```

However, while **foldl** unconditionally evaluates the spine you can still selectively evaluate the values in the list:

```
Prelude> foldl (\_ _ -> 5) 0 ([1..5] ++ undefined)
*** Exception: Prelude.undefined
-- error because bottom is part of the spine
-- and foldl must evaluate the spine

Prelude> foldl (\_ _ -> 5) 0 ([1..5] ++ [undefined])
5
-- this is OK because here bottom is a value
```

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 except it is strict. In other words, it forces evaluation of the values inside cons cells as it traverses the spine, rather than accumulating unevaluated expressions for each element of the 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` for the function. So 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 going to always be one of the elements in the list and `b` is either my 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 list `foldr` 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 actually 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 `String`:

```
foldr (\ a b -> take 3 a) "" ["Pizza", "Apple", "Banana"]
```

Now this will already typecheck and work, but it doesn't match the semantics we asked for:

```
Prelude> let pab = ["Pizza", "Apple", "Banana"]
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 did 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 actually folding the list. We're just 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 actually accumulate the result by making use of the `b` argument. Remember the `b` is your `acc` value. Technically we could just 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> concat $ map (take 3) (reverse pab)
"BanAppPiz"
```

But we need an excuse to play with `foldr` and `foldl`, so we'll pretend none of this happened!

```
Prelude> foldr (\ a b -> take 3 a ++ b) "" pab
"PizAppBan"
Prelude> foldl (\ b a -> take 3 a ++ b) "" pab
"BanAppPiz"
```

Here we concatenated the result of having taken three elements from the string value in our input result onto the front of our string we're accumulating. If we want to be explicit, we can assert types for the values:

```
Prelude> let f a b = take 3 (a :: String) ++ (b :: String)
Prelude> foldr f "" pab
"PizAppBan"
```

If we assert something that isn't true, the typechecker fwacks us:

```
Prelude> let f a b = take 3 (a :: String) ++ (b :: [String])
Prelude> foldr f "" pab

<interactive>:12:42:
    Couldn't match type 'Char' with '[Char]'
    Expected type: [String]
      Actual type: [Char]
    In the second argument of '(++)', namely '(b :: [String])'
    In the expression: take 3 (a :: String) ++ (b :: [String])
```

This can be useful for checking that your mental model of the code is accurate.

Intermission: Exercises

Write the following functions for processing this data.

```haskell
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))
  , 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.

   ```haskell
   filterDbDate :: [DatabaseItem] -> [UTCTime]
   filterDbDate = undefined
   ```

2. Write a function that filters for `DbNumber` values and returns a list of the `Integer` values inside them.

   ```haskell
   filterDbNumber :: [DatabaseItem] -> [Integer]
   filterDbNumber = undefined
   ```

3. Write a function that gets the most recent date.

   ```haskell
   mostRecent :: [DatabaseItem] -> UTCTime
   mostRecent = undefined
   ```

4. Write a function that sums all of the `DbNumber` values.

   ```haskell
   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> take 3 $ foldr (:) [] ([1, 2, 3] ++ undefined)
[1,2,3]
Prelude> take 3 $ foldl (flip (:)) [] ([1, 2, 3] ++ undefined)
*** 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 just 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> foldr const 0 (repeat 0 ++ [1,2,3])
0
Prelude> foldl (flip const) 0 (reverse (repeat 1 ++ [1,2,3]))
^CInterrupted.
-- ^^ bottom.
```

If we flip our folding function **f** and reverse the list **xs**, **foldr** and **foldl**
will return the same result:

```
Prelude> foldr (:) [] [1..5]
[1,2,3,4,5]
Prelude> foldl (flip (:)) [] [1..5]
[5,4,3,2,1]
Prelude> foldl (flip (:)) [] (reverse [1..5])
[1,2,3,4,5]
Prelude> reverse $ foldl (flip (:)) [] [1..5]
[1,2,3,4,5]
```

10.8 Summary

Okay, 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 passed 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 (tail-call) through the list, only beginning to produce values after it's reached 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 the 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 difference is 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 (folds *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

---

[3]The truth is scans are not used tremendously 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 of this, please see Chris Done's blog post about his solution to the waterfall problem at `http://chrisdone.com/posts/twitter-problem-loeb`.

and evaluation are similar. This does mean that you can use scans in places that 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]
[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 a bunch
= [1, 1 + 1, (1 + 1) + 2, ((1 + 1) + 2) + 3]

-- evaluating additions
= [1, 2, 4, 7]
```

Then to make this more explicit and properly equational, we can follow along with how **scanl** expands for this expression just 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 the Recursion chapter, we wrote a recursive function that returned the nth Fibonacci number to us. You can use a scan function to return a list

of Fibonacci numbers. We're going to do this in a source file because this
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 nonrecursive list of
numbers whose start value is 1 would just give us this:

```
scanl (+) 1 [1..3]
[(((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 only take some number of elements
from that list or to find the nth element as we had done 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 zero. That is, the first value in your list is indexed as
zero. 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`. We will 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:

```
*Main> fibsN 0
1
*Main> fibsN 2
2
*Main> 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 Recursion 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?

Prime number machine

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:

```
myFunc = foldr f z
```

So for example with the **and** function:

```haskell
-- Again, this type will be less reusable than
-- the one in GHC 7.10 and newer. Don't worry.

-- 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 in the folding function
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`.

   ```haskell
   myOr :: [Bool] -> Bool
   myOr = undefined
   ```

2. `myAny` returns `True` if `a -> Bool` applied to any of the values in the list returns `True`.

   ```haskell
   myAny :: (a -> Bool) -> [a] -> Bool
   myAny = undefined
   ```

   Example for validating `myAny`:

   ```haskell
   Prelude> myAny even [1, 3, 5]
   False
   Prelude> myAny odd [1, 3, 5]
   True
   ```

3. In addition to the recursive and fold based `myElem`, write a version that uses `any`.

   ```haskell
   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.

   ```haskell
   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`.

   ```haskell
   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 results.

```
squishMap :: (a -> [b]) -> [a] -> [b]
squishMap = undefined

Prelude> squishMap (\x -> [1, x, 3]) [2]
[1,2,3]
Prelude> squishMap (\x -> "WOOO " ++ [x] ++ " HOOOO ") "blah"
"WOOO b HOOOO WOOO l HOOOO WOOO a HOOOO WOOO h HOOOO "
```

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 returned `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 returned `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:

```haskell
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:

```haskell
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` is a function whose tail calls are recursive invocations of itself. This is distinguished from functions that call other functions in their tail call.

```
f x y z = h (subFunction x y z)
    where subFunction x y z = g x y z
-- Not tail recursive, 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 actual 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   Answers

First intermission exercises

1. `foldr (*) 1 [1..5]`

will return the same result as **b** and **c**. The first possible choice won't typecheck.

2.
```haskell
foldl (flip (*)) 1 [1..3]

foldl (flip (*)) 1 [1, 2, 3]

-- Keeping in mind
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f z0 xs0 = lgo z0 xs0
          where
              lgo z []     =  z
              lgo z (x:xs) = lgo (f z x) xs

foldl :: (b -> a -> b) -> b -> [a] -> b
foldl (flip (*)) 1 [1, 2, 3] = lgo 1 [1, 2, 3]
          where
              lgo z []     =  z
              lgo z (x:xs) = lgo (f z x) xs

foldl (flip (*)) 1 [1, 2, 3] = lgo 1 [1, 2, 3]
  where
    lgo z []     =  z
    lgo z (x:xs) = lgo (f z x) xs

-- lgo 1 [] -- didn't match
lgo 1 (1 : [2, 3]) =
  lgo ((flip (*)) 1 1) [2, 3]

-- lgo 1 [] -- didn't match
lgo 1 (2 : [3]) =
  lgo ((flip (*)) ((flip (*)) 1 1) 2) [3]
```

```
-- lgo 1 [] -- didn't match
lgo 1 (3 : []) =
  lgo ((flip (*)) ((flip (*)) ((flip (*)) 1 1) 2) 3) []

lgo 1 [] =
    ((flip (*)) ((flip (*)) ((flip (*)) 1 1) 2) 3)
-- lgo 1 (x : xs) -- didn't match

-- Cleaning it up a bit

((flip (*)) ((flip (*)) ((flip (*)) 1 1) 2) 3)

-- ((flip (*)) 1 1) is just 1 * 1

((flip (*)) ((flip (*)) (1 * 1) 2) 3)

-- ((flip (*)) (1 * 1) 2) is (2 * (1 * 1))

((flip (*)) (2 * (1 * 1)) 3)

-- same deal as before, put the 3 in front
-- of the nested expression.

(3 * (2 * (1 * 1)))

-- If we care to reduce it

(3 * (2 * (1 * 1)))
    (6 * (1 * 1))
         6
```

You can confirm this with the trick Cale Gibbard popularized. Note we flipped the x and the y to reflect that (*) was flipped in our expression.

```
Prelude> let xs = ["1", "2", "3"]
Prelude> let z = "1"
Prelude> foldl (\x y -> concat ["(",y,"*",x,")"]) z xs
"(3*(2*(1*1)))"
```

You can further convince yourself that Haskell will execute in the way the terms seem to evaluate on paper with the use of `undefined` as a bottom value.

```
Prelude> foldl (flip const) undefined [undefined, undefined, 3]
3
```

What the above has done is created an expression similar to the previous one, except it ignored all values but the 3 and returned that alone.

3. Answer not provided.

4. Catamorphisms are conventionally used to reduce structure, but they're not limited to that. Mapping, filtering, and the like are expressible in terms of a fold.

5. Answers not provided.

Second intermission exercises

1. Couple different approaches are possible for this one.

```haskell
filterDbDate :: [DatabaseItem] -> [UTCTime]
filterDbDate =
  foldr maybeCons []
  where maybeCons a b =
          case a of
            (DbDate date) -> date : b
            _ -> b
```

Could also have done.

```haskell
filterDbDate :: [DatabaseItem] -> [UTCTime]
filterDbDate = foldr (\a b -> filterHelper a ++ b) []

filterHelper :: DatabaseItem -> [UTCTime]
filterHelper (DbDate t) = [t]
filterHelper _ = []
```

Or even better (recursively concatenating lists is slow):

```haskell
-- don't forget to import catMaybes
import Data.Maybe (catMaybes)

getDate :: DatabaseItem -> Maybe UTCTime
getDate (DbDate t) = Just t
getDate _ = Nothing

filterDbDate'' :: [DatabaseItem] -> [UTCTime]
filterDbDate'' db = catMaybes xs
  where xs = fmap getDate db
```

Then if you wanted to factor out the "choice" of value.

```haskell
filterDb :: (DatabaseItem -> Maybe a)
         -> [DatabaseItem]
         -> [a]
filterDb getter db = catMaybes xs
  where xs = fmap getter db
```

Which can then be used in the following manner:

```haskell
Prelude> filterDb getDate theDatabase
[1911-05-01 09:28:43 UTC,1921-05-01 09:28:43 UTC]
```

We know you're not comfortable with the `Maybe` type yet, this a taster and this is a good chance to give you a peek of it. One uses this type when you either do or do not have a single value.

2. Answer not provided, should be awful similar to previous answer.

3. Answer not provided, only a hint that you've already played with functions that find the highest ordinal value of a list.

4. Answer not provided.

5. Answer not provided.

Scans Exercises

Answers not provided.

Chapter Exercises

Warm-up and review

1. a) There are many ways to write this one. You will not understand all of them, that's okay. It's also okay if you arrived at a different answer.

```haskell
import Control.Applicative

stops, vowels :: String
stops = "pbtdkg"
vowels = "aeiou"

combosDo :: [a] -> [b] -> [c] -> [(a, b, c)]
combosDo xs ys zs = do
  x <- xs
  y <- ys
  z <- zs
  return (x, y, z)

combosApply :: [a] -> [b] -> [c] -> [(a, b, c)]
combosApply xs ys zs =
  (,,) <$> xs <*> ys <*> zs

combosListComp :: [a] -> [b] -> [c] -> [(a, b, c)]
combosListComp xs ys zs =
  [(x, y ,z) | x <- xs, y <- ys, z <- zs]
```

```haskell
combosBind :: [a] -> [b] -> [c] -> [(a, b, c)]
combosBind xs ys zs =
  xs >>=
  (\x -> ys >>=
        (\y -> zs >>=
              (\z ->
                return (x, y, z))))
combosCM :: [a] -> [b] -> [c] -> [(a, b, c)]
combosCM xs ys zs =
  concatMap (\x ->
    concatMap (\y ->
      concatMap (\z ->
                  return (x, y, z))
      zs)
    ys)
  xs

combosFoldr :: [a] -> [b] -> [c] -> [(a, b, c)]
combosFoldr xs ys zs =
  foldr ((++) . (\x ->
    foldr ((++) . (\y ->
      foldr ((++) . (\z ->
                    return (x, y, z)))
      [] zs))
    [] ys))
  [] xs
```

b)
```haskell
fst' :: (a, b, c) -> a
fst' (a, _, _) = a

-- Rewrote the combosApply variant
combosPOnly :: String
            -> String
            -> String
            -> [(Char, Char, Char)]
combosPOnly xs ys zs =
  filter ((==) 'p' . fst') candidates
  where candidates = (,,) <$> xs <*> ys <*> zs
```

c) Answer not provided.

2. Answer not provided.

3. Answer not provided.

Rewriting functions using folds

1. `myOr = foldr (||) False`

2. Answer not provided.

3. Answer not provided.

4. `myReverse = foldl (flip (:)) []`

5. `myMap f = foldr ((:) . f) []`

6. Answer not provided.

7. `squish = foldr (++) []`

8. `squishMap f = foldr ((++) . f) []`

9. Answer not provided.

10. Answer not provided.

11. Answer not provided.

10.13 Follow-up resources

1. Haskell Wiki. Fold.
   `https://wiki.haskell.org/Fold`

2. Richard Bird. Sections 4.5 and 4.6 of Introduction to Functional Programming using Haskell (1998).

3. Antoni Diller. Introduction to Haskell
   `http://www.cantab.net/users/antoni.diller/haskell/units/unit06.html`

4. Graham Hutton. A tutorial on the universality and expressiveness of fold.
   `http://www.cs.nott.ac.uk/~gmh/fold.pdf`

Chapter 11

Algebraic datatypes

Control the types, control the universe

> 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

441

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 offers 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 usage of type synonyms and `newtype`;

- introduce *kinds.*

---

[1]This description, slightly edited for our purposes, was proposed by Orah Kittrell in the `#haskell-beginners` IRC channel

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 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 constructors 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. Equals sign divides the type constructor from its data constructors.

4. Data constructor. In this case, a data constructor that takes no arguments and so 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 what values can have that type.

6. Constructor for the value `True`, another nullary constructor.

7. Type constructor with an argument. An empty list has to have 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 a `[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 argument. But first, let's turn our attention to the meaning of the word *constructors*.

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 typeclass 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 value is 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 parametric. 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.

```haskell
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.[2]

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

---

[2]If you're interested in learning about this topic, Brent Yorgey's blog posts about type families and functional dependencies are a good place to start. `https://byorgey.wordpress.com/2010/06/29/typed-type-level-programming-in-haskell-part-i-functional-dependencies/`

value-level witnesses! Some are *phantom.* We will cover this in a later chapter.

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 list of whatever other concrete type – obviously the set of possible lists is large).

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.

We query the kind signature of a type constructor (not a data constructor) in GHCi with a `:kind` or `:k`. Compare the following:

```
Prelude> let f = not True
Prelude> :t f
f :: Bool

Prelude> let 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 just 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> let f x = x > 3
Prelude> :t f 5
f 5 :: Bool
```

And 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 is itself a concrete type. This is what the *constructor* of "type constructor" is referring to.

11.4   Data constructors and values

We mentioned just 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.

```haskell
data PugType = PugData
--   [1]         [2]

data HuskyType a = HuskyData
--       [3]           [4]

data DogueDeBordeaux doge = DogueDeBordeaux doge
--          [5]                     [6]
```

1. `PugType` is the type constructor, but it takes no arguments so we can think of it as being a type *constant* because it's really just a constant type. 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 *phantom*, or, "has no witness." We will elaborate on this later. Here `HuskyData` is a constant value just 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. The difference here is 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 – we know 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:

```haskell
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:

```haskell
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:

```
Prelude> :k Doggies
Doggies :: * -> *
-- this needs to be applied to become a
-- concrete type

Prelude> :t Husky
Husky :: a -> Doggies a
-- this needs to be applied to become a
-- concrete value
```

So the behavior of our 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.

Intermission: Exercises

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!"`?

11.5   What's a type and what's data?

As we've 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 does not persist through to runtime. Data are what we're working with at runtime.

Here 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 what 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]

-- type constructor (a)
-- data constructor (b)
-- 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` wasn'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)

-- one type constructor (c)
-- three data constructors (d), (e), and (f)
```

```haskell
data Airline =
--      (g)
        PapuAir
--        (h)
      | CatapultsR'Us
--         (i)
      | TakeYourChancesUnited
--         (j)
        deriving (Eq, Show)

-- one type constructor (g).
-- 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:

```haskell
data Vehicle = Car Manufacturer Price
--    (k)      (l)     [2]        [3]
             | Plane Airline
--             (m)      [4]
             deriving (Eq, Show)

-- type constructor (k)
-- two data constructors (l) and (m).
-- three type arguments [2], [3], and [4]
-- two type arguments to (l) are [2] and [3]
-- type argument to (m) is [4]
```

In the above, the datatypes are generating 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 it is almost always true that you will want to derive an instance

of `Show` for any datatype 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 typeclasses that allow derivation in this manner, and it obviates the need for manually writing instances for each datatype and typeclass (reminder: you saw an example of this in the Typeclasses chapter).

As we've seen, data constructors can take arguments. Those arguments will be specific types, but not specific values. In standard Haskell, we cannot quotient out or choose specific inhabitants (values) of types as arguments. You can't say, "`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`.

Intermission: Exercises

For these exercises, we'll use the datatypes defined in the above section. It would be good if you'd typed them all into a source file already, but if you hadn't please do so now. You can then define some sample data on your own, or use these to get you started:

```haskell
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:

```haskell
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:

```haskell
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've decided 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.

11.6   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 which 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 which stand for themselves and act as a witness of the datatype they were declared in.

We've said that "A type can be thought of as an enumeration of constructors that have zero or *more* arguments," so it stands to reason that not all data constructors are nullary. You have seen examples of data constructors that take one or more arguments, but we haven't made too much of it yet.

Earlier in this chapter, we saw how data constructors may take arguments and that makes them more like a function 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:

```haskell
-- 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> Example2 10 "Flappity Bat" == Example2 10 "NotCom"
False
```

Our `Example2` is an example of a *product*. Another example of a data constructor that takes multiple arguments is the tuple, which can take several arguments – as many as there are inhabitants of each tuple – and

for that reason, tuples can be called *anonymous products.* We'll talk more about product types soon.

Unary (one argument) data constructors contain a single value of whatever type their argument was. 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 deriving (Eq, Show)
--   [1]       [2]   [3] [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 definition of the data constructor from [2].

4. Deriving clause.

5. Typeclass instances being derived. We're getting equality `Eq` and value stringification `Show` for free.

```
Prelude> data MyType = MyVal Int deriving (Eq, Show)
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 – only one – `Int` value.

11.7   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.[3] 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 which can generate a set.

The cardinality of a datatype is the number of possible values it defines. That number can be as small as 1 or as large as infinite (for example, numeric datatypes, 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 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, just because it has the small-

---

[3]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 cardinality.

est set of possible inhabitants and 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 the `maxBound` and `minBound` functions from the `Bounded` typeclass to view the 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.

Intermission: Exercises

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 – you can probably intuitively grasp what the cardinality is based just on what you know.

1. `data PugType = PugData`

2. For this one, recall that `Bool` is also defined with the `|`:

   ```
   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. Nullary constructors represent *one* value when reasoning 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, so 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.

Intermission: Exercises

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 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 typeclass 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 type constructor takes one argument. That argument will be a type, not a value. Now, instead of your constructor being a constant, or a known value, the value will be constructed at runtime from the type argument, as we demonstrated earlier.

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.

newtype

Having given considerable attention to unary data constructors, 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 defintions 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:

```haskell
tooManyGoats :: Int -> Bool
tooManyGoats n = n > 42
```

We might run into a problem here if we had different limits for different sorts of livestock. What if we mixed up the `Int` value from Cows where we meant Goats? Fortunately, there's a way to address this with unary constructors:

```haskell
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
```

Now we can't mix up our livestock counts:

```haskell
*Main Data.Char> tooManyGoats (Goats 43)
```

```
True
*Main Data.Char> 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)
```

Using **newtype** can deliver other advantages related to typeclass 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 gone 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 code, the distinction can be helpful in tracking where data came 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 typeclass instances for **newtype**s that differ from the instances for their underlying type. You can't do that for type synonyms. Let's take a look at how that works. We'll first define a typeclass called **TooMany** and an instance for **Int**:

```haskell
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:

```
*Main> 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 you wanted a special instance of `TooMany` that will have different behavior from the `Int` instance. Under the hood, `Goats` is still `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 were able to make the `Goats` newtype have an instance of `TooMany` which had different behavior than the type `Int` which it contains. We can't do this if it's just a type synonym. Don't believe us? Try it.

On the other hand, what about the case where we want to reuse the typeclass instances of the type our newtype contains? For common typeclasses 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 typeclasses, we can use a language pragma called "GeneralizedNewtypeDeriving". Language pragmas, also called extensions, are special instructions to the compiler. They tell the compiler to process input in ways beyond what the standard provides for. In this case, this pragma will tell the compiler to allow our `newtype` to rely on a typeclass 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 the special instruction to allow us to do this.

First, let's take the case of what we must do without generalized newtype deriving:

```haskell
class TooMany a where
  tooMany :: a -> Bool

instance TooMany Int where
  tooMany n = n > 42

newtype Goats = Goats Int deriving (Eq, Show)

-- this will do the same thing as the
-- Int instance, but we still have to
-- define it separately

instance TooMany Goats where
  tooMany (Goats n) = tooMany n
```

You can test this yourself to see that they'll return the same answers.

Now we'll add the pragma at the top of our source file:

```haskell
{-# 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)
```

Now we don't have to define an instance of `TooMany` for `Goats` that's merely identical to the `Int` instance. We can reuse the instance that we already have.

This is also nice for times when we want every typeclass instance to be the same *except* for the one we want to change.

Intermission: Exercises

1. Reusing the `TooMany` typeclass, write an instance of the typeclass 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 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.

11.8 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*. We've seen sum types previously, such as `Bool`:

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

We've 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:

---

[4] `https://ghc.haskell.org/trac/haskell-prime/wiki/FlexibleInstances`

```haskell
-- How many values inhabit Bool?
data Bool = False | True

-- okay lets drop the type constructor for now
-- ?? stands in for our cardinality
True | False = ??

-- Given that |, the sum type syntax, is + or addition
True + False == ??

-- and that False and True both == 1
1 + 1 == ??

-- We see that the cardinality of Bool is
1 + 1 == 2

-- List of all possible counts for Bool
[True, False] -- length is 2
```

From this, we see that when working with a `Bool` value we must reason about two possible values. Sum types are a way of expressing alternate 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 as we saw.

Intermission: Exercises

1. Given a datatype

    ```haskell
    data BigSmall =
        Big Bool
      | Small Bool
      deriving (Eq, Show)
    ```

What is the cardinality of this datatype? Hint: We already know **Bool**'s cardinality. Show your work as demonstrated earlier.

2. Given a datatype

```
-- needed to have Int8 in scope
import Data.Int

data NumberOrBool =
    Numba Int8
  | BoolyBool Bool
  deriving (Eq, Show)

-- Example use of Numba, parentheses due to
-- syntactic collision between (-) minus and
-- the negate function
let 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> let n = Numba (-128)
Literal 128 is out of the Int8 range -128..127
If you are trying to write a large negative
  literal, use NegativeLiterals
```

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 maxBound 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 the warning is the following:

```
Prelude> let n = (-128)
Prelude> let x = Numba n
```

Or you can use the NegativeLiterals extension as it recommends:

```
Prelude> :set -XNegativeLiterals
Prelude> let n = Numba (-128)
```

Note that the negative literals extension doesn't prevent the warning if you use `negate`.

11.9   Product types

What does it mean for a type to be a product? A product type's cardinality is the *product* of the cardinality of its inhabitants. Arithmetically, products are *multiplication*. Where a sum type was expressing *or*, a product type expresses *and*.

For those that have programmed in C-like languages before, a product is like a struct. For those that 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.

If that seems confusing, think for a moment about tuples. We said previously that tuples are anonymous products. The declaration of the tuple type looks like this:

```
( , ) :: a -> b -> (a, b)
```

This is a product, like a product type: it gives you a way to encapsulate two pieces of data, of possibly (though not necessarily) different types, in a single value.

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 will become 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:

```
MkTwoQs QuantumTrue QuantumTrue
MkTwoQs QuantumTrue QuantumFalse
MkTwoQs QuantumTrue QuantumBoth
MkTwoQs QuantumFalse QuantumFalse
-- ...... and so on
```

Note that there is no special syntax denoting product types as there was 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 `QuantumBool`s. 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 have also 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 the record. Let's begin by definining a simple product type:

```haskell
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 the value we want from our little box of values:

```haskell
-- these are just 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 just the `String` value from your data.

Now let's see how we could define a similar product type but with record syntax:

```haskell
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 ProductType -> 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> let 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.

Intermission: Jammin Exercises

Here we've started working on datatypes to keep track of Julie's homemade jam output, with an `Int` value to represent how many jars she's canned:

```
data Fruit =
    Peach
  | Plum
  | Apple
  | Blackberry
  deriving (Eq, Show)

data JamJars =
  Jam Fruit Int
  deriving (Eq, Show)
```

1. Let's make a module for this. Name your module at the top of the file:

   module Jammin **where**

2. Rewrite `JamJars` with record syntax.

3. What is the cardinality of `JamJars`?

4. Add `Ord` instances to your deriving clauses.

5. You can use the record field accessors in other functions as well. To demonstrate this, work up some sample data that has a count of the types and numbers of jars of jam in the rows in our pantry (you can define more data than this if you like):

   ```
   row1 = undefined
   row2 = undefined
   row3 = undefined
   row4 = undefined
   row5 = undefined
   row6 = undefined
   allJam = [row1, row2, row3, row4, row5, row6]
   ```

   Now over that list of data, we can map the field accessor for the `Int` value and see a list of the numbers for each row.

6. Write a function that will return the total number of jars of jam.

7. Write a function that will tell you which row has the most jars of jam in it. It should return a result like this, though the fruit and number will vary depending on how you defined your data:

   ```
   *Jammin> mostRow
   Jam {fruit = Apple, jars = 10}
   ```

8. Under your module name, import the module called `Data.List`. It includes some standard functions called `sortBy` and `groupBy` that will allow us to organize our list of jams. Look at their type signatures because there are some important differences between them.

9. You'll want to sort the list **allJams** by the first field in each record. You may (or may not) want to use the following helper function as part of that:

```
compareKind (Jam k _) (Jam k' _) = compare k k'
```

10. Now take the sorting function and use **groupBy** to group the jams by the type of fruit they are made from. You'll later want the ability to sum the sublists separately, so you're looking for a result that is a list of lists (again, the actual data in your list will depend on how you defined it):

```
*Jammin> groupJam
 [ [ Jam {fruit = Peach, jars = 5}
   , Jam {fruit = Peach, jars = 3} ]
 , [ Jam {fruit = Plum, jars = 8}
   , Jam {fruit = Plum, jars = 4} ]
 , [ Jam {fruit = Apple, jars = 10} ]
 , [ Jam {fruit = Blackberry, jars = 7}
   , Jam {fruit = Blackberry, jars = 4} ] ]
```

Save this module as we will work more with it in a later chapter.

11.10   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:

```
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:

```haskell
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 but only permit 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 permit 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`. If you'd like, 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 broke those values out and made 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 very common to papers about type systems and programming languages:

```haskell
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) + . . .

A stricter interpretation of normal form or "sum of products" would require representing products with tuples and sums with Either. The previous datatype in that form would look like the following:

```haskell
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 use for beginners.

The `Either` type will be explained in detail in the next chapter.

Exercises

1. Given the type

```haskell
data FlowerType = Gardenia
                | Daisy
                | Rose
                | Lilac
                deriving Show

type Gardener = String

data Garden =
  Garden Gardener FlowerType
  deriving Show
```

What is the normal form of `Garden`?

11.11 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 talked 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.

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:

```haskell
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

Here Sum and Product are ways to represent arbitrary sums and products in our types. In ordinary Haskell code it's unlikely you'd need or want nestable sums and products unless you were doing something fairly advanced, but here we use them as a means of demonstration.

If we have just 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 the following examples):

```haskell
-- if the counts could overflow,
-- then the farm can afford the
-- programmer time to convert
-- the system

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 the compiler chokes:

```haskell
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`:

```haskell
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 a finer wool.

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)

-- Alternately

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`:

```
-- Getting it right
Prelude> let bess = First (CowInfo "Bess" 4) :: Animal'

Prelude> let elmer' = Second (SheepInfo "Elmer" 5 5)
Prelude> let elmer = Second elmer' :: Animal'

-- Making a mistake
Prelude> let elmo' = Second (SheepInfo "Elmo" 5 5)
Prelude> let elmo = First elmo' :: Animal'

Couldn't match expected type 'CowInfo'
            with actual type 'Sum a0 SheepInfo'
In the first argument of 'First', namely
  '(Second (SheepInfo "Elmo" 5 5))'
In the expression:
  First (Second (SheepInfo "Elmo" 5 5)) :: 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> :t First (Second (SheepInfo "Baaaaa" 5 5))
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 it can be useful to develop an intuition
about this structure to 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. Nothing special in the syntax here. We just define `trivialValue` to be the nullary data constructor `Chickenbutt` and we have a value of the 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:

```
-- note:
-- MkId :: a -> Id a

idInt :: Id Integer
idInt = MkId 10
```

As we've said throughout the book, one of the functional parts of functional programming is that functions themselves are merely values. So we can also do this:

```
idIdentity :: Id (a -> a)
idIdentity = MkId $ \x -> x
```

This is a little odd. The type `Id` takes an argument and the data constructor `MkId` takes an argument of the corresponding polymorphic type. So, in order to have a value of type `Id Integer`, we need to apply `a -> Id a` to an `Integer` value. This binds the *a* type variable to `Integer` and applies away the `(->)` in the type constructor, giving us `Id Integer`. We can also construct a `MkId` value that is an identity function by binding the *a* to a polymorphic function in both the type and the term level.

Moving along, 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 just 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'd defined above. The `Product` data constructor is a function of two arguments, the `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> Second Twitter :: Sum Twitter AskFm

Couldn't match expected type 'AskFm' with
actual type 'Twitter'

In the first argument of 'Second', namely 'Twitter'
In the expression:
  Second Twitter :: Sum Twitter AskFm

Prelude> First AskFm :: Sum Twitter AskFm

Couldn't match expected type 'Twitter' with
actual type 'AskFm'

In the first argument of 'First', namely 'AskFm'
In the expression:
```

```
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 in a datatype
that is written differently like so:

```
data SocialNetwork =
    Twitter
  | AskFm
  deriving (Eq, Show)
```

Now the data constructors for `Twitter` and `AskFm` are direct inhabitants
of the sum type `SocialNetwork`, where before they inhabited the `Sum` type.
Now let's consider how this might look with type synonyms:

```
type Twitter = String
type AskFm = String

twitter :: Sum Twitter AskFm
twitter = First "Twitter"

-- It has no way of knowing
-- we made a mistake because
-- both values are just Strings
askfm :: Sum Twitter AskFm
askfm = First "Twitter"
```

There's a problem with the above example. The name of `askfm` implies we
meant `Second "AskFm"`, but we messed up. Because we used type synonyms
instead of defining datatypes, the type system didn't swat us for it! 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:

```
>>> :t RecordProduct
RecordProduct :: a -> b -> RecordProduct a b

>>> :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 defined 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:

```haskell
data OperatingSystem =
      GnuPlusLinux
    | OpenBSDPlusNevermindJustBSDStill
    | Mac
    | Windows
    deriving (Eq, Show)

data ProgrammingLanguage =
      Haskell
    | Agda
    | Idris
    | PureScript
    deriving (Eq, Show)

data Programmer =
  Programmer { os :: OperatingSystem
             , lang :: ProgrammingLanguage }
  deriving (Eq, Show)
```

Then we can construct a value from the record product `Programmer`:

```
Prelude> :t Programmer
Programmer :: OperatingSystem
           -> ProgrammingLanguage
           -> Programmer
```

```haskell
nineToFive :: Programmer
nineToFive = Programmer { os = Mac
                        , lang = Haskell }

-- We can reorder stuff when we use record syntax
feelingWizardly :: Programmer
feelingWizardly = Programmer { lang = Agda
                            , os = GnuPlusLinux }
```

Exercise

Write a function that generates all possible values of `Programmer`. Use the provided lists of inhabitants of `OperatingSystem` and `ProgrammingLanguage`.

```haskell
allOperatingSystems :: [OperatingSystem]
allOperatingSystems =
  [ GnuPlusLinux
  , OpenBSDPlusNevermindJustBSDStill
  , Mac
  , Windows
  ]

allLanguages :: [ProgrammingLanguage]
allLanguages = [Haskell, Agda, Idris, PureScript]

allProgrammers :: [Programmer]
allProgrammers = undefined
```

Since `Programmer` is a product of `OperatingSystem` and `ProgrammingLanguage`, you can determine how many inhabitants of `Programmer` you have by calculating:

```haskell
length allOperatingSystems * length allLanguages
```

This is the essence of how product types and the number of inhabitants relate.

If after running `nub` from Data.List to remove duplicate values over your allProgrammers value, it 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> let partialAf = Programmer {os = GnuPlusLinux}

    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:

```haskell
-- Works the same as if we'd used record syntax.

data ThereYet =
  There Integer Float String Bool
  deriving (Eq, Show)

-- who needs a "builder pattern"?
nope :: Float -> String -> Bool -> ThereYet
nope = There 10

notYet :: String -> Bool -> ThereYet
notYet = nope 25.5

notQuite :: Bool -> ThereYet
notQuite = notYet "woohoo"

yusssss :: ThereYet
yusssss = notQuite False

-- Not I, said the Haskell user.
```

Notice the way our types progressed.

```haskell
There :: Integer -> Float -> String -> Bool -> ThereYet
nope ::             Float -> String -> Bool -> ThereYet
notYet ::                    String -> Bool -> ThereYet
notQuite ::                            Bool -> ThereYet
yusssss ::                                     ThereYet
```

Percolate values through your programs, not bottoms.[5]

Deconstructing values

When we discussed folds, we mentioned the idea of catamorphism.[6] We explained that catamorphism was about *deconstructing* lists. This idea

---

[5]A favorite snack of the North American Yeti is bottom-propagating Haskellers

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:

```haskell
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 ole 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:

```haskell
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.

Now an alternate formulation with a product that uses record syntax:

---

[6]Om nom nom nom

```haskell
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.

Dang it, more 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:

```haskell
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 one has a `Null` value, but you've used one of the record accessors:

```
Prelude> make Null
"*** Exception: No match in record selector make

-- Don't.
```

How do we fix this? Well, first, whenever we have a product that uses record accessors, keep it separate of 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:

```haskell
-- 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 Haskell, we want the typechecker to catch us doing things wrong, so we can fix it *before* problems multiply and things go wrong at runtime. But the typechecker can best help those who help themselves.

11.12 Function type is exponential

In the arithmetic of calculating inhabitants of types, 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 arithmetic laws, 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` is kinda unfortunate: 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:

```haskell
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:

```haskell
-- 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

quantFlip6 :: Quantum -> Quantum
quantFlip6 Yes  = No
quantFlip6 No   = Yes
quantFlip6 Both = Yes
```

```haskell
quantFlip7 :: Quantum -> Quantum
quantFlip7 Yes  = Both
quantFlip7 No   = Yes
quantFlip7 Both = Yes

quantFlip8 :: Quantum -> Quantum
quantFlip8 Yes  = Both
quantFlip8 No   = Yes
quantFlip8 Both = No

quantFlip9 :: Quantum -> Quantum
quantFlip9 Yes  = Both
quantFlip9 No   = No
quantFlip9 Both = No

quantFlip10 :: Quantum -> Quantum
quantFlip10 Yes  = Both
quantFlip10 No   = No
quantFlip10 Both = Both

-- You can figure out the remaining
-- possibilities yourself.
```

Exponentiation in what order?

Consider the following function:

```haskell
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.

Intermission: Exercises

Determine how many unique inhabitants each type has.

Suggestion: just do the arithmetic unless you want to verify. Writing them out gets tedious quickly.

1. ```haskell
   data Quad =
       One
     | Two
     | Three
     | Four
     deriving (Eq, Show)

   -- how many different forms can this take?
   eQuad :: Either Quad Quad
   eQuad = ???
   ```

2. ```haskell
   prodQuad :: (Quad, Quad)
   ```

3. ```haskell
   funcQuad :: Quad -> Quad
   ```

4. ```haskell
   prodTBool :: (Bool, Bool, Bool)
   ```

5. ```haskell
   gTwo :: Bool -> Bool -> Bool
   ```

6. Hint: 5 digit number

   ```haskell
   fTwo :: Bool -> Quad -> Quad
   ```

11.13 Higher-kinded datatypes

You may recall 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.

```
-- Silly polymorphic product type
-- this is 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 the
author maintains called Bloodhound.[7]

```
data EsResultFound a =
  EsResultFound {  _version :: DocVersion
                , _source  :: a
  } deriving (Eq, Show)
```

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` typeclass 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` typeclass 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` typeclass 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.

---

[7]`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 notation for transmitting
data, especially between servers and web applications.

11.14 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:

```
-- Haskell definition of lists aka []
data []   a  =   []  |  a    :    [a]
--   [1] [2]      [3]   [4] [5]  [6]
```

1. Type constructor for Haskell list, special `[]` syntax

2. Single type argument to `[]`, this is the type of value our list contains

3. Nil / empty list, again special [] syntax. [] marks the end of the list.

4. A single value of type `a`

5. `:` is an infix constructor. It is a product of `a` [4] and `[a]` [6]

6. The rest of our 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 this 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.

The list type in Prelude has already taken the lone colon as the data con-
structor for the cons cells, but you can add non-alphanumeric characters to

get your own unique infix type or data constructor. Here's an some example of an infix data constructor:

```haskell
data Product a b =
    a :&: b
    deriving (Eq, Show)
```

Then using it in the REPL:

```
Prelude> 1 :&: 2
1 :&: 2
Prelude> :t 1 :&: 2
1 :&: 2 :: (Num a, Num b) => Product a b
```

A value of type Product would be a product of two arguments, one of type *a* and one of type *b*.

Whether or not you choose to use infix data constructors, type constructors, or typeclass names is down to aesthetic preference.

In the following example, we'll define the list type without using an infix constructor:

```haskell
-- Same type, but redefined with slightly 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 / empty list, marks end of list as well.

4. A single value of type `a` in the `Cons` product

5. `Cons` constructor, product of `a` and `List a`

6. The rest of our list

How do we use our List type?

```
-- from GHCi
Prelude> let nil = Nil

-- the type parameter isn't applied because
-- Nil by itself doesn't tell the type inference
-- what the List contains.
Prelude> :t nil
nil :: List a

Prelude> let 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 type constructs `List` and `[]` before we've applied their 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.

11.15   Binary Tree

Now 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 actually 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:

```haskell
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.

```haskell
-- Leaf being our "empty tree" case

Prelude> let t1 = insert' 0 Leaf
Prelude> t1
Node Leaf 0 Leaf
```

```
Prelude> let t2 = insert' 3 t1
Prelude> t2
Node Leaf 0 (Node Leaf 3 Leaf)

Prelude> let 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 merely 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. We know this seems bewildering, but if you can believe it, 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. Just write the recursive functions and get it done.

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 (just like when you are mapping functions over lists).

```
-- filling in some details to help you along
-- 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 okay!"
  else error "test failed!"

-- hints for implementing mapTree below this code block
```

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 ignored 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`

Now the `Node` return needs to have a value of type b and `BinaryTree` values with type b inside them. You have two functions at your disposal. One gets you `(a -> b)`, the other maps `BinaryTree`s of type a into `BinaryTree`s of type b. Get 'em tiger.

A few suggestions that might help you with this exercises.

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 at 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.

```haskell
preorder :: BinaryTree a -> [a]
preorder = undefined

inorder :: BinaryTree a -> [a]
inorder = undefined

postorder :: Ord a => 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 "postorder failed check"

main :: IO ()
main = do
  testPreorder
  testInorder
  testPostorder
```

Write foldr for BinaryTree

Given the definition of `BinaryTree` we have provided, write a catamorphism
for the binary trees.

```
-- any traversal order is fine
foldTree :: (a -> b -> b -> b) -> b -> BinaryTree a -> b
```

Rewrite map for BinaryTree

Using the `foldTree` you just wrote, rewrite `mapTree` using `foldTree`. The
absence of an Ord constraint is intentional, you don't need to use the insert
function.

```
mapTree' :: (a -> b) -> BinaryTree a -> BinaryTree b
mapTree' f bt = foldTree ...?
```

11.16 Chapter Exercises

Multiple choice

1. Given the following datatype:

    ```
    data Weekday =
        Monday
      | Tuesday
      | Wednesday
      | Thursday
      | Friday
    ```

    we can say:

    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. and 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) delivers the head of `xs`

    c) delivers the final element of `xs`

    d) has the same type as `xs`

Ciphers

In the Lists chapter, 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 plaintext 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
```

Now 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, so "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 the cipher 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:

```haskell
f :: Show a => (a, b) -> IO (a, b)
f t@(a, _) = do
  print a
  return t
```

Here we pattern-matched on a tuple so we could get at the first value for printing, but used 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 construc-
tors, which includes lists:

```haskell
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 in implementing 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.

   ```haskell
   isSubsequenceOf :: (Eq a) => [a] -> [a] -> Bool
   ```

   The following are examples of how this function should work:

   ```
   Prelude> isSubsequenceOf "blah" "blahwoot"
   True
   Prelude> isSubsequenceOf "blah" "wootblah"
   True
   Prelude> isSubsequenceOf "blah" "wboloath"
   True
   Prelude> isSubsequenceOf "blah" "wootbla"
   False
   ```

2. Split a sentence into words, then tuple each word with the capitalized
   form of each.

```
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 "Titter"
"Titter"
Prelude> capitalizeWord "titter"
"Titter"
```

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> capitalizeParagraph "blah. woot ha."
"Blah. Woot ha."
```

Phone exercise

This exercise by geophf[8] originally for 1HaskellADay.[9] Thank you for letting us use this exercise!

Remember the days when you had a cell phone and you had to press the digit 7 four times to get the letter 'S' to show up in your text that your texting with your friendies?

Yeah: that.

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 + _    |   # .,      |
-----------------------------------------
```

Where star (*) gives you capitalization of the letter you're writing to your friendies. Oh, and 0 is the space bar. And to represent the digit itself, you press that digit once more than the letters it represents.

Isn't this quaint?

We're going to kick this around.

1. Create a data structure that captures the phone layout above.

   ```haskell
   data DaPhone = Sommat
   ```

2. High school kids, ya know? They like texting to each other sweet nothings. Convert the following conversations into the keypresses required to express them.

```haskell
convo :: [String]
convo = ["Wanna play 20 questions",
         "Ya",
         "U 1st haha",
         "Lol ok. Have u ever tasted alcohol lol",
         "Lol ya",
         "Wow ur cool haha. Ur turn",
         "Ok. Do u think I am pretty Lol",
         "Lol ya",
         "Haha thanks just making sure rofl ur turn"]

-- validButtons = "1234567890*#"
type Digit = Char

-- valid presses = [1..4]
type Presses = Int

cellPhonesDead :: DaPhone -> String -> [(Digit, Presses)]
cellPhonesDead = undefined
```

3. How many times do digits need to be pressed for each message?

```haskell
fingerTaps :: [(Digit, Presses)] -> Presses
fingerTaps = undefined
```

4. What was the most popular letter for each message? What was its cost? You'll want to combine `reverseTaps` and `fingerTaps` to figure out what it cost in taps. `reverseTaps` is a list because you need to press a different button in order to get capitals.

```haskell
popularest :: String -> Char
popularest = undefined

reverseTaps :: Char -> [(Digit, Presses)]
reverseTaps = undefined
-- reverseTaps 'a' == ('2', 1)
-- reverseTaps 'A' == [('*', 1), ('2', 1)]
```

5. What was the most popular letter overall? What was the most popular word?

```haskell
coolestLtr :: [String] -> Char
coolestLtr = undefined

coolestWord :: [String] -> String
coolestWord = undefined
```

Hutton's Razor

Hutton's Razor[10] is a very simple expression language that expresses integer literals and addition of values in that expression language. 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 which reduces an expression to a final sum.

   ```haskell
   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.

   ```haskell
   printExpr :: Expr -> String
   printExpr = undefined
   ```

---

[10]http://www.cs.nott.ac.uk/~pszgmh/bib.html#semantics

Expected output:

```
Prelude> printExpr (Add (Lit 1) (Lit 9001))
"1 + 9001"
Prelude> let a1 = Add (Lit 9001) (Lit 1)
Prelude> let a2 = Add a1 (Lit 20001)
Prelude> let a3 = Add (Lit 1) a2
Prelude> printExpr a3
"1 + 9001 + 1 + 20001"
```

11.17   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.

11.18   Answers

Data constructors and values

1. Type constructor

2. Answer is right above where the exercises are.

3. In the case of `Doggies String`, the `* -> *` has been applied, so it now has kind `*`.

4. `Num a => Doggies a`

5. Answer not provided.

6. `Doggies String` or `Doggies [Char]`, String is just a type alias for `[Char]` anyway.

7. Somewhat unfortunately, the answer is "both" and you're going to see this a lot in Haskell code where the type constructor and sole data constructor for a type have the same name. They're still independent entities in Haskell.

8. This question is about the data constructor, so it's asking for the type of the function that constructs a value of that type. The answer is readily determined in the REPL and we demonstrated it just above.

9. Answer not provided.

What's a type and what's data?

1. Answer not provided.

2. We'll give you one of the functions.

   ```
   isCar :: Vehicle -> Bool
   isCar (Car _ _) = True
   isCar _ = False
   ```

3. Answer not provided.

4. Kaboom. Don't write partial functions like this.

5. Answer not provided.

What makes these datatypes algebraic?

1. `PugType` has a cardinality of 1, there is only one inhabitant: the lone `PugData` data constructor.

2. `Airline` has a cardinality of 3. There are three data constructors.

3. 65536.

   ```
   Prelude> import Data.Int
   Prelude> maxBound :: Int16
   32767
   Prelude> minBound :: Int16
   -32768
   Prelude> 32768 * 2
   65536
   ```

   The `maxBound` is 32767 because you need to account for 0 as well. Computers usually store an `Int16` type in 2 bytes of memory.

4. Answer not provided, but consider that Integer is not finite.

5. Computers are implicitly using a binary counting system. You can actually replicate this on your fingers and get a more efficient counting system than what most westerners do. Most westerners will count using a unary system, 1 finger up means 1, 2 fingers up means 2, and so forth. You can encode more information than that with your fingers if you treat a finger being down or up as two discrete bits of information. That shifts it from being unary to binary.

In computers, we usually measure things in bytes. A byte is 8 bits. Accordingly, if you count with your fingers in binary, 8 of your fingers can represent a single byte. We can use cardinality calculations to determine how many distinct values that can represent. Here we exponentiate to get the answer. The base is the discrete number of values a "digit" can represent, the exponent is the number of digits you have. In a binary system, the base is always 2. If you have a byte, you have 8 bits, so the cardinality of a byte is 2^8.

We already know that 10 fingers can represent 0 through 10 in the unary counting system. However, if you count in binary, you'll be able to count up to 2^10. 10 bits of information lets us represent 1,024 discrete values.

How does this look on hands? Consider this textual approximation. Each | is a finger that is raised, each _ is a finger that is lowered.

```
-- I usually count with my palms
-- facing me, thumbs outward.

t: thumb
i: index finger
m: middle finger
r: ring finger
p: pinky

   t i m r p   p r m i t
0: _ _ _ _ _   _ _ _ _ _
1: _ _ _ _ _   _ _ _ _ |
2: _ _ _ _ _   _ _ _ | _
3: _ _ _ _ _   _ _ _ | |
4: _ _ _ _ _   _ _ | _ _
```

```
 5:  _ _ _ _ _    _ _ | _ |
 6:  _ _ _ _ _    _ _ | | _
 7:  _ _ _ _ _    _ _ | | |
 8:  _ _ _ _ _    _ | _ _ _
 9:  _ _ _ _ _    _ | _ _ |
10:  _ _ _ _ _    _ | _ | _
11:  _ _ _ _ _    _ | _ | |
12:  _ _ _ _ _    _ | | _ _
13:  _ _ _ _ _    _ | | _ |
14:  _ _ _ _ _    _ | | | _
15:  _ _ _ _ _    _ | | | |
```

In the previous example, we've counted from 0 to 15, which is actually 16 discrete values because we're choosing to count from 0.

If you dig up a binary to decimal converter on the web and type in a binary number like 1101100100, you'll get 868 back. Note how the ones and zeroes line up with your fingers.

```
      1 1 0 1 1   0 0 1 0 0
868: | | _ | |   _ | | | |
```

Anyhoo, Int8 is so-called because it's 8 bits and 2^8 equals 256.

Simple datatypes with nullary data constructors

No answers provided.

newtype

Possible answers for all three in the following module. There is room to have written instances that did different things. This wasn't a principled application of typeclasses, so do not consider this exemplary.

```
{-# LANGUAGE FlexibleInstances #-}

module TooMany where

class TooMany a where
  tooMany :: a -> Bool

instance TooMany Int where
  tooMany n = n > 42

instance TooMany String where
  tooMany s = length s > 42

-- suffices for (Int, Int) as well
instance (Num a, TooMany a) => TooMany (a, a) where
  tooMany (a, a') = tooMany (a + a')
```

Sum types

1. ```
   data BigSmall = Big Bool | Small Bool

   -- Bool = 2
   data BigSmall = Big 2 | Small 2

   -- Sum is addition for the purposes of cardinality
   data BigSmall = Big 2 + Small 2

   -- Unary constructor takes on the
   -- cardinality of its contents.
   data BigSmall = 2 + 2

   -- Cardinality of BigSmall is 4.
   ```

2. Answer not provided.

Intermission: Jammin Exercises

Answers not provided.

Normal form

```haskell
type Gardener = String

data Garden =
    Gardenia Gardener
  | Daisy Gardener
  | Rose Gardener
  | Lilac Gardener
  deriving Show
```

Alternately:

```haskell
type Gardener = String

data Gardenia = Gardenia Gardener
data Daisy = Daisy Gardener
data Rose = Rose Gardener
data Lilac = Lilac Gardener

type Garden =
  Either Gardenia
    (Either Daisy (Either Rose Lilac))
```

Constructing and deconstructing values

No answer provided.

Function type is exponential

Intermission: Exercises

1. 4 nullary data constructors, so 4. Either is the anonymous sum, and sums are addition, so it's Quad + Quad, which is $4 + 4 = 8$.

2. Answer not provided.

3. Answer not provided.

4. Answer not provided.

5. 16, 2 to the power of 2 to the power of 2.

6. No answer, but remember that it's b^a for a function `a -> b`.

Lists are polymorphic

Write map for tree

We're not rebalancing, so we can keep this one simple.

```
mapTree :: (a -> b) -> BinaryTree a -> BinaryTree b
mapTree _ Leaf = Leaf
mapTree f (Node left a right) =
  Node (mapTree f left) (f a) (mapTree f right)
```

Chapter Exercises

Multiple choice

1. Answer not provided.

2. Answer not provided.

3. b

4. Answer not provided.

As-patterns

Answers not provided.

Language exercises

1.
```haskell
capitalizeWord :: String -> String
capitalizeWord w
    | null w = w
    | otherwise = [toUpper firstLetter] ++ map toLower others
    where ([firstLetter], others) = splitAt 1 w
```

2. Answer not provided.

Phone exercise

Answers not provided.

Hutton's Razor

Answers not provided.

Chapter 12

Signaling adversity with Maybe and Either

> Thank goodness we don't have
> only serious problems, but
> ridiculous ones as well
>
> _Edsger W. Djikstra_

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 received 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 used in a lot of Haskell programs.

This chapter will include:

- Nothing, or Just Maybe

- Either left or right, but not both

- higher-kindedness

- anamorphisms, but not animorphs.

12.2   How I learned to stop worrying and love Nothing

Let's consider the definition of Maybe again:

```haskell
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 datatype.

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 happened because the number wasn't even:

```haskell
ifEvenAdd2 :: Integer -> Integer
ifEvenAdd2 n = if even n then n+2 else ???
```

What can we do to make it say, "hey, this number wasn't even so I have nothing for you, my friend?" Well, instead of promising an `Integer` result, we can return `Maybe Integer`:

```haskell
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 the type `Integer`. We need to wrap that value in the other constructor `Maybe` offers: `Just`. Here's the error you'd get if you tried to load it:

```
<interactive>:9:75:
    Couldn't match expected type 'Maybe Integer'
                with actual type 'Integer'
    In the first argument of '(+)', namely 'n'
    In the expression: n + 2
```

And here's how we fix it:

```haskell
ifEvenAdd2 :: Integer -> Maybe Integer
ifEvenAdd2 n = if even n then Just (n+2) else Nothing
```

We had to parenthesize `n+2` because function application binds the most tightly in Haskell (has the highest precedence), so the compiler otherwise would've parsed 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 which keeps track of two things, their name and their age. We'll write this up as a simple product type:

```
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 we do not.

This is much better than our original, but what if we want to know if it was the name, age, or both that was bad? We may want to tell our user something was wrong with their input. Fortunately, we have a datatype for that!

12.3   Bleating either

So now we want a way to express why we didn'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 were incorrect *if* they were incorrect. So we'll start by making a simple sum type to enumerate our failure modes.

```
data PersonInvalid = NameEmpty
                   | AgeTooLow
                   deriving (Eq, Show)
```

By now, you know why we derived `Show`, but it's important that we derive `Eq` 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

-- Compiles fine without Eq
toString :: PersonInvalid -> String
toString NameEmpty = "NameEmpty"
toString AgeTooLow = "AgeTooLow"

instance Show PersonInvalid where
  show = toString

-- This does not work without an Eq instance
blah :: PersonInvalid -> String
blah pi
  | pi == NameEmpty = "NameEmpty"
  | pi == AgeTooLow = "AgeTooLow"
  | otherwise = "???"
```

It's worth considering that if you needed to have an Eq instance to pattern
match, how would you write the Eq instances?

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:

```haskell
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` 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

   ```haskell
   name /= "" && age >= 0 = Right (Person name age)
   ```

   instead of using the dollar sign.

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 a stop to whatever work is being done by your program. You can't apply functions to the left type argument, and `functor` will not map over it (you may remember `Functor` from our introduction of `fmap` back in the chapter about 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 really a person.[1]

```
Prelude> :t mkPerson "Djali" 5
mkPerson "Djali" 5 :: 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" 0
Left AgeTooLow
Prelude> mkPerson "" 0
Left NameEmpty
```

Notice in the last example that when both the name and the age are wrong, we're just 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'll change is that instead of validating all the data for a `Person` at once, we're going to make separate checking

---

[1]Don't know what we mean? Check the name Djali on a search engine.

functions and then combine the results. We'll see means of abstracting patterns like this out later:

```haskell
type Name = String
type Age = Integer
type ValidatePerson a = Either [PersonInvalid] a
-- this type alias wasn't in our previous version
-- otherwise, these types are the same as above

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:

```haskell
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" is still going to return as a valid name. Try it.

- If you try to put an Integer in for the name, you won't get a `Left` result, you'll get a type error. Try it. Similarly 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 `ValidatePerson` type.

3. Our main function now relies on a similarly-named helper function.

4. First argument to this function is the result of the `nameOkay` function.

5. 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'` are just plain old pattern matches.

Now let's see what we get:

```
Prelude> mkPerson "" 0
Left [NameEmpty,AgeTooLow]
```

Ahh, that's more like it. Now we can tell the user what was incorrect in one go without them having to round-trip each mistake with our program! What's really going to bake your noodle is that we'll be able to replace `mkPerson` and `mkPerson'` with the following later in this book:

```
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*. Here 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 types. In the Report, *type constructor* is used to refer to types which must have arguments applied to become a 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 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 | RoofGoats | Woot a
```

`Example` is a type constructor rather than a constant because it takes a type argument *a* which is used with the `Woot` data constructor. In GHCi we can query kinds with `:k`:

```
Prelude> data Example a = Blah | RoofGoats | Woot a
Prelude> :k Example
Example :: * -> *
```

`Example` has one argument, so it must be applied to one type in order to become a concrete type represented by a single `*`. The two-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 are concrete types. Just as we saw with the effect of currying in type signatures, applying `Maybe` to an `a` type constructor relieves us of one arrow and makes it a kind 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 `#` and 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 type which *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 are unlifted

because their representation is identical to that of the type they contain, so the 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?

```
Prelude> newtype 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 "real" 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'
The first argument of 'Maybe' should have kind '*',
  but 'Maybe' has kind '* -> *'
In a type in a GHCi command: 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 in fact 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:

```
Prelude> data Example a = Blah | RoofGoats | Woot a
```

```
Prelude> :k Maybe Example
```

```
Expecting one more argument to 'Example'
The first argument of 'Maybe' should have kind '*',
  but 'Example' has kind '* -> *'
In a type in a GHCi command: 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 some nice looking 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 just a `Maybe []` for the same reason we couldn't have just
a `Maybe Maybe`, but we can have a `Maybe [Bool]`:

```
Prelude> :k Maybe []


Expecting one more argument to '[]'
The first argument of 'Maybe' should have kind '*',
  but '[]' has kind '* -> *'
In a type in a GHCi command: Maybe []
Prelude> : k Maybe [Bool]

<interactive>:1:1: Not in scope: type variable 'k'
Prelude> :k Maybe [Bool]
Maybe [Bool] :: *
```

If you recall, one of the first times we used `Maybe` in the book was to write
a "safe" version of a `tail` function back in the chapter on Lists:

```
safeTail        :: [a] -> Maybe [a]
safeTail []     = Nothing
safeTail (x:[]) = Nothing
safeTail (_:xs) = Just xs
```

As soon as we apply this to a value, the polymorphic type variables become constrained or concrete types:

```
*Main> safeTail "julie"
Just "ulie"
*Main> :t safeTail "julie"
safeTail "julie" :: Maybe [Char]

*Main> safeTail [1..10]
Just [2,3,4,5,6,7,8,9,10]
*Main> :t safeTail [1..10]
safeTail [1..10] :: (Num a, Enum a) => Maybe [a]

*Main> :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:

```
Prelude> data Trivial = Trivial
Prelude> :k Trivial
Trivial :: *

Prelude> data Unary a = Unary a
Prelude> :k Unary
Unary :: * -> *

Prelude> data TwoArgs a b = TwoArgs a b
Prelude> :k TwoArgs
TwoArgs :: * -> * -> *
```

```
Prelude> data ThreeArgs a b c = ThreeArgs a b c
Prelude> :k ThreeArgs
ThreeArgs :: * -> * -> * -> *
```

It may not be clear why this is useful to know right now, other than helping to understand when your type errors are caused by things not being fully applied. The implications of higher-kindedness will be clearer in a later chapter.

Data constructors are functions

In the previous chapter, we noted the difference between data constants and data constructors and noted that data constructors that haven't been fully applied have a function arrow in them. Once you apply them to their argument, they return a value of the appropriate type. In other words, data constructors really are functions. As it happens, they behave just like Haskell functions in that they are curried as well.

First let's observe that nullary data constructors that are really just values and take no arguments are *not* like functions:

```
Prelude> 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:

```
Prelude> data UnaryC = UnaryC Int deriving Show
Prelude> :t UnaryC
UnaryC :: Int -> UnaryC
Prelude> UnaryC 10
UnaryC 10
Prelude> :t UnaryC 10
```

```
UnaryC 10 :: UnaryC
```

And just 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]'
```

If we wanted a unary data constructor which could contain any type, we would parameterize the type like so:

```
Prelude> 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 an Show instance, it won't cause a problem until we try to show the value:

```
-- no problem with using the id function
Prelude> :t (Unary id)
(Unary id) :: Unary (t -> t)

-- but id doesn't have a Show instance
Prelude> show (Unary id)

<interactive>:53:1:
    No instance for (Show (t0 -> t0))
     ...
```

The only way to avoid this would be to write an instance that did not show that value along with Unary, 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:

```
Prelude> data Unary = Unary a deriving Show

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 more clear 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 linguist co-authors enjoy doing in their spare time.

1. Write a recursive function that takes a text/string, breaks it into words and replaces each instance of "the" with "a". It's intended only to replace exactly the word "the".

```
-- example GHCi session above the functions

-- >>> notThe "the"
-- Nothing
-- >>> notThe "blahtheblah"
-- Just "blahtheblah"
-- >>> notThe "woot"
-- Just "woot"
notThe :: String -> Maybe String
notThe = undefined

-- >>> replaceThe "the cow loves us"
-- "a cow loves us"
replaceThe :: String -> String
replaceThe = undefined
```

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 "the cow"
-- 0
-- >>> countTheBeforeVowel "the evil cow"
-- 1
countTheBeforeVowel :: String -> Integer
countTheBeforeVowel = undefined
```

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 vowelhood

   b) Return the vowels of a string

   c) Count the number of elements returned

```
-- >>> countVowels "the cow"
-- 2
-- >>> countVowels "Mikolajczak"
-- 4
countVowels :: String -> Integer
countVowels = undefined
```

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 indicates the input isn't a real 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 Naturals to Integers and Integers to Naturals. The conversion from Naturals to Integers won't return Maybe because Integers are a strict superset of Naturals. Any Natural can be represented by an Integer, but the same is *not* true of any Integer. Negative numbers are not valid natural numbers.

```haskell
-- As natural as any competitive bodybuilder
data Nat =
    Zero
  | Succ Nat
  deriving (Eq, Show)

-- >>> natToInteger Zero
-- 0
-- >>> natToInteger (Succ Zero)
-- 1
-- >>> natToInteger (Succ (Succ Zero))
-- 2
natToInteger :: Nat -> Integer
natToInteger = undefined

-- >>> integerToNat 0
-- Just Zero
-- >>> integerToNat 1
-- Just (Succ Zero)
-- >>> integerToNat 2
-- Just (Succ (Succ Zero))
-- >>> integerToNat (-1)
-- Nothing
integerToNat :: Integer -> Maybe Nat
integerToNat = undefined
```

Small library for Maybe

Write the following functions. This may take some time.

1. Simple boolean checks for Maybe values.

```
-- >>> isJust (Just 1)
-- True
-- >>> isJust Nothing
-- False
isJust :: Maybe a -> Bool

-- >>> isNothing (Just 1)
-- False
-- >>> isNothing Nothing
-- True
isNothing :: Maybe a -> Bool
```

2. The following is the `Maybe` catamorphism. You can turn a `Maybe` value
   into anything else with this.

```
-- >>> mayybee 0 (+1) Nothing
-- 0
-- >>> mayybee 0 (+1) (Just 1)
-- 2
mayybee :: b -> (a -> b) -> Maybe a -> b
```

3. In case you just want to provide a fallback value.

```
-- >>> fromMaybe 0 Nothing
-- 0
-- >>> fromMaybe 0 (Just 1)
-- 1
fromMaybe :: a -> Maybe a -> a

-- Try writing it in terms of the maybe catamorphism
```

4. Converting between `List` and `Maybe`.

```
-- >>> listToMaybe [1, 2, 3]
-- Just 1
-- >>> listToMaybe []
-- Nothing
listToMaybe :: [a] -> Maybe a

-- >>> maybeToList (Just 1)
-- [1]
-- >>> maybeToList Nothing
-- []
maybeToList :: Maybe a -> [a]
```

5. For when we just want to drop the `Nothing` values from our list.

```
-- >>> catMaybes [Just 1, Nothing, Just 2]
-- [1, 2]
-- >>> catMaybes [Nothing, Nothing, Nothing]
-- []
catMaybes :: [Maybe a] -> [a]
```

6. You'll see this called "sequence" later.

```
-- >>> flipMaybe [Just 1, Just 2, Just 3]
-- Just [1, 2, 3]
-- >>> flipMaybe [Just 1, Nothing, Just 3]
-- Nothing
flipMaybe :: [Maybe a] -> Maybe [a]
```

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, just 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 very limited
-- unfold that never ends
Prelude> :t iterate
iterate :: (a -> a) -> a -> [a]

-- because it never ends, 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 the full monty
Prelude> :t unfoldr
unfoldr :: (b -> Maybe (a, b)) -> b -> [a]

-- Using unfoldr to do the same as thing as iterate
Prelude> take 10 $ unfoldr (\b -> Just (b, b+1)) 0
[0,1,2,3,4,5,6,7,8,9]
```

Why bother?

We bother with this for the same reason we abstracted 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?

```haskell
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 (++) []
```

Your eyes may be spouting gouts of blood, 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.

   ```haskell
   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.

   ```haskell
   myUnfoldr :: (b -> Maybe (a, b)) -> b -> [a]
   myUnfoldr = undefined
   ```

3. Rewrite `myIterate` into `betterIterate` using `myUnfoldr`. A hint – we used `unfoldr` to produce the same results as `iterate` earlier. 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 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've just 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.

Chapter 13

Building projects in Haskell

> 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, typeclasses, typeclass instances, and values you've defined 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 languages 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. Haskell is sufficiently different from other languages that we think it's best to spend time getting pretty comfortable with how Haskell itself works before trying to build substantial projects. What most often happens is the student realizes how much they still don't understand about Haskell, shakes their fist at the sky, and curses Haskell's very name and all the elitist jerks who write Haskell and flees to relative safety. Nobody wants that.

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, and work with modules as they are in Haskell. 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.

Just try to relax and have fun with this. You've earned it after those binary tree exercises.

In this chapter, we'll cover:

- building Haskell programs with modules;
- using the Cabal package manager;

- conventions around Haskell project organization;

- building a small interactive game.

13.2   Managing projects with Cabal

The Haskell Cabal, or Common Architecture for Building Applications and Libraries, is a package and dependency manager. "Package" and "dependency" refer to the program you've built along with the interlinked elements of that program, which can include the code you write, the modules you import, tests you have associated with the project, and documentation of the project. Cabal exists to help organize all this and make sure all dependencies are properly in scope. It also builds the project, that is, it turns all this into a program (or library – more on that later) you can use.

While the Haskell community does not have a prescribed project layout, we recommend the following basic structure:

- Make a `src` directory for source code.

- Make a file called `Main.hs` that is the main Haskell source code file.

- Use `cabal init` to automatically generate the configuration and licensing files.

- Initialize git for version control.

- Your project directory will also contain the test suites and any other files that your `Main` module depends on.

We're going to start learning Cabal by building a sample project called `hello-haskell`. First, make a directory for this project called `hello-haskell`. Once you are working within that directory, do a `cabal init`. It's going to start asking you questions that will automatically generate some files for you within that directory. You can choose the default setting for most of them simply by hitting return, and you can always access the `.cabal` file it will generate to change setting when you need to.

When you reach this question:

```
What does the package build:
   1) Library
   2) Executable
```

You want to choose Executable rather than Library.

At this question, you'll notice that the default is, as we said above, for the main module of the program to be `Main.hs`

```
What is the main module of the executable:
 * 1) Main.hs (does not yet exist)
   2) Main.lhs (does not yet exist)
   3) Other (specify)
```

Don't worry that it does not yet exist; it will exist soon enough.

This question allows you to include explanations of what each field in your `.cabal` file means. You may find it useful to include that when you are learning Cabal:

```
Include documentation on what each field means (y/n)? [default: n]
```

Once you've gone through the list of questions, you'll see something like this:

```
Guessing dependencies...

Generating LICENSE...
Generating Setup.hs...
Generating hello-haskell.cabal...
```

And then you'll have a little directcory with some files in it. At this point, it should look something like this:

```
$ tree
```

```
.
├── hello-haskell.cabal
├── LICENSE
└── Setup.hs
```

You may want to edit the `.cabal` file at this point and add a Description field or edit any fields you want changed.

Alternative cabal init You can also initialize cabal non-interactively:

```
$ cabal init -n -l BSD3 --is-executable \
--language=Haskell2010 -u bitemyapp.com \
-a 'Chris Allen' -c User-interface \
-s 'Saying Hello' -p hello-haskell
```

Edit the above, of course, as appropriate for your project.

Version control While we do recommend that you use version control, such as git, for your projects, we're not going to include a lot of git instructions. Version control is important, and if you're unfamiliar with how to use it, we do recommend learning sooner rather than later, but that is mostly outside the scope of this book. If you are new to programming, this little section might be difficult-to-impossible to follow, so come back to it later.

We recommend adding the `.gitignore` from Github's gitignore repository plus some additions for Haskell so we don't accidentally check in unnecessary build artifacts or other things inessential to the project. This should go into a file named `.gitignore` at the top level of your project. Visit `https://github.com/github/gitignore` and get copy the contents of the Haskell.gitignore file into your project's .gitignore file.

With the `.gitignore` file in place, you can add the files you have so far to your git repo's staging and then commit it to the history:

```
$ git add .
$ git commit -m "Initial commit"
```

You might be wondering why we're telling `git` to ignore something called a "cabal sandbox" in the `.gitignore` file. Cabal, unlike the package managers in other language ecosystems, requires direct and transitive dependencies to have compatible versions. For contrast, Maven will use the closest version. To avoid packages having conflicts, Cabal introduced sandboxes which let you do builds of your projects in a way that doesn't use your user package-db. Your user package-db is global to all your builds on your user account and this is almost never what you want. This is not dissimilar from `virtualenv` in the Python community. The `.cabal-sandbox` directory is where our build artifacts will go when we build our project or test cases. We don't want to version control that as it would bloat the git repository and doesn't need to be version controlled.

Haddock   Haddock is a tool for automatically generating documentation from annotated Haskell source code and is the most common tool used in the Haskell community for generating documentation. It's most often used for documenting libraries, although it can be used with any kind of Haskell code. We will not be covering it in detail in this chapter.

Examples of good project layout   As you start wanting to build longer, more complex projects, you may want to study and follow some well-designed project layouts. Edward Kmett's lens library[1] is not only a fantastic library in its own right but also a great resource for people wanting to see how to structure a Haskell project, write and generate Haddock[2] documentation, and organize namespaces. Kmett's library follows Hackage guidelines[3] on what namespaces and categories to use for his libraries.

An alternative namespacing pattern is demonstrated by Pipes, a streaming library.[4] It uses a top-level eponymous namespace. For an example of another popular project you could also look at Pandoc[5] for examples of how to organize non-trivial Haskell projects.

---

[1] lens library Github repository `https://github.com/ekmett/lens`

[2] Haddock website `https://www.haskell.org/haddock/`

[3] Hackage guidelines `https://wiki.haskell.org/Package_versioning_policy`

[4] Pipes hackage page `http://hackage.haskell.org/package/pipes`

[5] Pandoc github repository `https://github.com/jgm/pandoc/`

A sidebar about executables and libraries

When we chose "Executable" in the initialization of our project, it's because we're making a command-line application which will be run and used. When we're writing code we want people to be able to reuse in other projects, we make a library and choose which modules we want to expose. Where 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, executables are applications that the operating system will run directly.

If you'd like to read further on this, here are a few links.

1. `https://en.wikipedia.org/wiki/Static_library`

2. `https://en.wikipedia.org/wiki/Executable`

Editing the Cabal file

Let's get back to editing that `cabal` file a bit. Ours is named `hello-haskell.cabal` and is in the top level directory of the project.

Right now, your `cabal` file should look something like :

```
-- Initial hello-haskell.cabal generated by cabal init.
-- For further documentation,
-- see http://haskell.org/cabal/users-guide/

name:               hello-haskell
version:            0.1.0.0
synopsis:           blah blah
-- description:
license:            BSD3
license-file:       LICENSE
author:             (your name)
maintainer:         (your email address)
-- copyright:
-- category:
```

```
build-type:          Simple
-- extra-source-files:
cabal-version:       >=1.10

executable hello
  main-is:             Main.hs
  -- other-modules:
  -- other-extensions:
  build-depends:       base >=4.7 && <4.8
  -- hs-source-dirs:
  default-language:    Haskell2010
```

The named executable stanza allows you to build a binary by that name and run it. You'll need to edit the following:

1. Set the description so Cabal will stop squawking about it. Cabal has a synopsis and a description for two different purposes. Synopsis is a brief description for a tabular listing of libraries. Description is freeform and can be as long as you like. In practice, most Haskellers will put a fuller description and explanation of the library in a `README` file which is markdown formatted, usually with the file extension `md` so that the entire filename is `README.md`.

2. Set `hs-source-dirs` to `src` so Cabal knows where our modules are. The two hyphens in front of `hs-source-dirs` make it a comment, so remove the hyphens and add `src` in the same column as `Main.hs` and `base >=4.7 && <4.8` etc.

3. Ensure `main-is` is set to `Main.hs` in the executable stanza so the compiler knows what main function to use for that binary.

4. Add `ghc-options: -Wall -fwarn-tabs` to the executable stanza so we get the *very* handy warnings GHC offers on top of the usual type checking.

5. Add the libraries our project will use to `build-depends`. For now, we only have `base`, which is the basic Haskell package that includes Prelude and its supporting libraries. We'll need to add other dependencies in later projects, but this will do for now.

Building and running your program

The next step in building a project is to create the `src` directory and then create a file named `Main.hs` in `src`.

For the purposes of our current toy project, the contents of `src/Main.hs` will be:

```
module Main where

main :: IO ()
main = putStrLn "Hello from Haskell!"
```

One thing to note is that for a module to work as a `main-is` target for GHC, it must have a function named `main` and itself be named `Main`. The `main` function must be a computation of type `IO a` where *a* is a type, often unit `()`. Executing the program calls the `main` function.

The `main` function in nontrivial programs is often contained within a wrapper module called `Main` which only exists to satisfy this purpose. Sometimes argument parsing and handling is done via libraries like optparse-applicative. [6]

For our toy project, though, we have only one module, and we've left it very simple, making it just a 'putStrLn' of the string '"Hello"'. To validate that everything is working, let's build and run this program.

We're going to create a Cabal sandbox so that our dependencies are isolated to this project.

```
$ cabal sandbox init
```

Once we've finished laying out our project, it's going to look like this:

```
$ tree
```

---

[6]optparse-applicative on Github `https://github.com/pcapriotti/optparse-applicative`

```
.
├── LICENSE
├── Setup.hs
├── cabal.sandbox.config
├── hello-haskell.cabal
├── src
│   ├── Main.hs
4 directories, 7 files
```

Now that our sandbox has been created, we want to install our dependencies into the Cabal sandbox package-db:

```
$ cabal install --only-dependencies
```

If you install the dependencies before you have your sandbox set up, they would be installed into the user package-db located in your home directory, which would be global to all the projects on your current user account. Installing dependencies can take some time on first run.

Now we're going to build our project:

```
$ cabal build
```

If this succeeds, we should get a binary named 'hello-haskell' in 'dist/build/hello-haskell'. To run this, do the following:

```
$ ./dist/build/hello-haskell/hello-haskell
Hello from Haskell!
$
```

You can also use `cabal repl` to open a REPL and call `main` like this:

```
$ cabal repl
*Main> main
Hello from Haskell!
```

Note that using `cabal repl` will compile the module as part of opening the REPL, and you can use the same commands you use in your GHCi, such as `:t` to get type information and `:q` to quit.

If everything is in place, let's move on to factoring out some code and making a separate module that can be exposed as a library.

13.3   Making our own modules

Let's back up just a minute and talk about what a module even is and how they are structured. As we've said, Haskell programs consist of a collection of modules. Modules serve the purpose of controlling the namespaces and are essentially big declarations that contain a variety of other declarations: data and type declarations, class and instance declarations, type signatures, function definitions, and so forth. A "namespace" is simply a set of objects (functions, etc), organized and contained in such a way that they may be referred to by name, without interference from other objects (functions, etc) that share the same name.

The namespace of modules is flat, and other than the `module` header and the import declarations, declarations may appear in any order.

A module always begins with a header with the keyword `module`, as we've shown you a few times. The module name may be followed by a list of entities that will be exported from that module, such as this list from a module called `Morse` that will be in the next chapter:

```haskell
module Morse
       ( Morse
       , charToMorse
       , morseToChar
       , stringToMorse
       , letterToMorse
       , morseToLetter
       ) where
```

The header is followed by a list of imports, which we will talk more about

shortly. The list of imports can be empty, of course, if the code within that module doesn't depend on any other libraries or modules.

We want to refactor our code for our `Hello` module a bit and look at an example of how to organize functions into modules, then reuse those functions inside other modules.

First, we're going to move our function that greets users out to its own module:

```
-- src/Hello.hs

module Hello where

sayHello :: IO ()
sayHello = putStrLn "Hello from Haskell!"
```

Then we're going to change our `Main.hs` to import the `Hello` module and use `sayHello` instead of its own code:

```
-- src/Main.hs

module Main where

import Hello

main = sayHello
```

Now we rebuild and run our program!

```
$ cabal build
Resolving dependencies...
Configuring hello-haskell-0.1.0.0...
Building hello-haskell-0.1.0.0...
Preprocessing executable 'hello-haskell'
for hello-haskell-0.1.0.0...
```

```
[1 of 2] Compiling Hello
[2 of 2] Compiling Main

src/Main.hs:5:1: Warning:
    Top-level binding with no type signature: main :: IO ()
Linking dist/build/hello-haskell/hello-haskell ...
$ dist/build/hello-haskell/hello-haskell
Hello from Haskell!
```

It works fine, but you may have seen that Warning message if you have
`-Wall` turned on in your configuration. We can fix that pesky warning
about our `main` function not having a type signature by adding the type
GHC has helpfully told us:

```
module Main where

import Hello

main :: IO ()
main = sayHello
```

Then we re-run our build:

```
$ cabal build
Building hello-haskell-0.1.0.0...
Preprocessing executable 'hello-haskell'
for hello-haskell-0.1.0.0...

[2 of 2] Compiling Main
Linking dist/build/hello-haskell/hello-haskell ...
$
```

No warning this time!

13.4 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 global scope throughout the 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've 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 what functions are included in the named module, while importing the module allows us to actually 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 used an unqualified import of everything in `Data.Bool`. What if we only wanted `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 pragma, or language 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 enter our REPL:

```
$ ghci -XNoImplicitPrelude
GHCi, version 7.8.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
>
```

We can check that **bool** and **not** are not in scope yet:

```
> :t bool

<interactive>:1:1: Not in scope: 'bool'
> :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**:

```
> import Data.Bool (bool)
Data.Bool> :t bool
bool :: a -> a -> GHC.Types.Bool -> a
Data.Bool> :t not

<interactive>:1:1: Not in scope: 'not'
```

Now, normally in the standard 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 your import declarations in your own module, just as we just demonstrated with GHCi. Putting `import Data.Char (toUpper)` in the import declarations of a module will ensure that `toUpper` is in scope for your project, but not any of the other entities contained in `Data.Char`.

For the examples in the next section, you'll want Prelude back on, so please restart GHCi before proceeding.

Qualified imports

What if you wanted to know *where* something you imported came 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. Here's an example of how you might use a qualified import:

```
Prelude> import qualified Data.Bool
Prelude> :t bool

<interactive>:1:1:
    Not in scope: 'bool'
    Perhaps you meant 'Data.Bool.bool' (imported from Data.Bool)

Prelude> :t Data.Bool.bool
Data.Bool.bool :: a -> a -> Data.Bool.Bool -> a

Prelude> :t Data.Bool.not
Data.Bool.not :: Data.Bool.Bool -> Data.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. Now we are marking where the functions that we're using came 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

<interactive>:1:1:
    Not in scope: 'bool'
    Perhaps you meant 'B.bool' (imported from Data.Bool)

Prelude> :t B.bool
B.bool :: a -> a -> B.Bool -> a

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

You can do qualified imports in the import declarations at the beginning of your module in the same way.

Intermission: Check your understanding

Here is the import list from one of the modules in Chris's library 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 `import Database.Blacktip.Types`
   is importing?

4. Now let's compare a small part of Blacktip's code to the above import
   list:

```haskell
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 did `forever` come from?

13.5   Making our program interactive

Now we're going to make our program ask for your name, then greet you by name. First, we're going to rewrite our `sayHello` function to take an argument:

```haskell
-- src/Hello.hs
sayHello :: String -> IO ()
sayHello name = putStrLn ("Hi " ++ name ++ "!")
```

Note we parenthesized the appending (++) function of the `String` argument to `putStrLn`.

Next we'll change `main` to get the user's name:

```haskell
-- src/Main.hs

main :: IO ()
main = do
  name <- getLine
  sayHello name
```

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 in a convenient syntax. Let's decompose what's going on here:

```haskell
main :: IO ()
main = do
--      [1]
   name  <-  getLine
-- [4]  [3]    [2]
   sayHello name
--           [5]
```

1. The **do** here begins the block.

2. **getLine** has type **IO String**, because it must perform IO (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 bound it to the variable **name**. Remember, **getLine** has type **IO String**, name has type **String**.

5. **sayHello** expects an argument **String**, which is the type of **name** but *not* **getLine**.

Now we'll fire off a build:

```
$ cabal build
Building hello-haskell-0.1.0.0...
Preprocessing executable 'hello-haskell'
for hello-haskell-0.1.0.0...

[1 of 2] Compiling Hello
[2 of 2] Compiling Main
Linking dist/build/hello-haskell/hello-haskell ...
$
```

And run the program:

```
$ dist/build/hello-haskell/hello-haskell
```

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.

What if we tried to pass getLine to sayHello?   If we tried to write
`main` without the use of `do` syntax, particularly without using `<-` such as in
the following example:

```haskell
main :: IO ()
main = sayHello getLine
```

We'll get the following type error:

```
$ cabal build
Building hello-haskell-0.1.0.0...
Preprocessing executable 'hello-haskell'
for hello-haskell-0.1.0.0...

[2 of 2] Compiling Main

src/Main.hs:8:17:
```

```
Couldn't match type 'IO String' with '[Char]'
Expected type: String
  Actual type: IO String
In the first argument of 'sayHello', namely 'getLine'
In the expression: 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 the 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 just need to change `main`:

```
-- src/Main.hs

module Main where

import System.IO

import Hello

main :: IO ()
main = do
  hSetBuffering stdout NoBuffering
  putStr "Please input your name: "
  name <- getLine
  sayHello name
```

We did several things here. One is that we used `putStr` instead of `putStrLn` so that our input could be on the same line as our prompt. We also imported from `System.IO` so that we could 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:

```
$ dist/build/hello/hello
Please input your name: julie
Hi julie!
```

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.

13.6 Using the REPL with a Cabal project

As we saw above, you can fire up GHCi via the `cabal` command and get a REPL that will be able to import your modules and any dependencies you might be using:

```
$ cabal repl
Preprocessing executable 'hello-haskell'
for hello-haskell-0.1.0.0...

GHCi, version 7.8.4: http://www.haskell.org/ghc/
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
[1 of 2] Compiling Hello
[2 of 2] Compiling Main
Ok, modules loaded: Main, Hello.
Prelude> import Hello
Prelude> sayHello "blah"
Hi blah!
Prelude>
```

`cabal repl` will by default choose a library if one exists. Otherwise, it will choose the first executable listed in your **cabal** file. You can also specify a target. The following **cabal** executable stanza has the target name **hello-haskell**:

```
executable hello-haskell
  ghc-options:       -Wall
  hs-source-dirs:    src
  main-is:           Main.hs
  build-depends:     base >= 4.7 && <5
  default-language:  Haskell2010
```

So you can specify this if you want as an argument to `cabal repl`:

```
$ cabal repl hello-haskell
...
[1 of 2] Compiling Hello
[2 of 2] Compiling Main
Ok, modules loaded: Main, Hello.
*Main> main
Please input your name: chica
Hi chica!
```

This can come in handy as you work on refactoring and expanding your modules.

13.7   Refactoring into a library

We could hypothetically use our **sayHello** function as part of a library by exposing the **Hello** module as part of the library and have our executable depend on that library. We'll need to change our Cabal file accordingly:

```
name:           hello
version:        0.1.0.0
```

```
synopsis:          Say hello
description:        Say hello
homepage:           bitemyapp.com
license:            BSD3
license-file:       LICENSE
author:             Chris Allen
maintainer:         cma@bitemyapp.com
copyright:          2014, Chris Allen
category:           Data
build-type:         Simple
cabal-version:      >=1.10

library
  exposed-modules:      Hello
  ghc-options:          -Wall
  hs-source-dirs:       src
  build-depends:        base >= 4.7 && <5
  default-language:     Haskell2010

executable hello-haskell
  ghc-options:          -Wall
  hs-source-dirs:       src
  main-is:              Main.hs
  build-depends:        base >= 4.7 && <5
                      , hello
  default-language:     Haskell2010
```

Note that when you target the library stanza in the Cabal file with `cabal repl`, `Main` is no longer visible, just `Hello`. The library stanza of a package is the default target, as you can see here:

```
$ cabal repl
...
Ok, modules loaded: Hello.
Prelude> import Main

<no location info>:
```

```
Could not find module 'Main'
It is not a module in the current
program, or in any known package.
```

The name of your library target in Cabal is whatever the overall name of the project is in the **name** setting you see at the top of our Cabal file. We named the executable from **hello-haskell** to avoid the following problem if they had been both named **hello**:

```
$ cabal repl hello
./hello.cabal has been changed.
Re-configuring with most recently used
options. If this fails, please run configure manually.

Resolving dependencies...
Configuring hello-0.1.0.0...
cabal: Ambiguous build target 'hello'. It could be:
exe:hello (component)
lib:hello (component)
```

Then if you want to be able to call **main** or otherwise experiment with the executable, you specify the name of the Cabal target you want to load. Note here how **Hello** and **Main** are listed in the modules loaded:

```
$ cabal repl hello-haskell
...
Ok, modules loaded: Hello, Main.
Prelude> main
Please input your name: blah
Hi blah!
Prelude>
```

13.8   Loading code from another project

The most straightforward and least troublesome way to manage project dependencies is to use sandboxes. Let's see how to use a local project as a dependency in another project.

Start in the directory above the `hello-haskell` project:

```
$ mkdir call-em-up
$ cd ./call-em-up
$ cabal init -n -l BSD3 --is-executable \
--language=Haskell2010 -u bitemyapp.com \
-a 'Chris Allen' -c User-interface \
-s 'Call the sayHello function' -p call-em-up
```

Then we edit our Cabal file:

```
name:           call-em-up
version:        0.1.0.0
synopsis:       Call the sayHello function
homepage:       bitemyapp.com
license:        BSD3
license-file:   LICENSE
author:         Chris Allen
maintainer:     cma@bitemyapp.com
category:       User-interface
build-type:     Simple
cabal-version:  >=1.10

executable call-em-up
  main-is:           Main.hs
  build-depends:     base >=4.7 && <5
                   , hello
  default-language:  Haskell2010
```

Note that we say our build depends on having `hello`.

We put a `Main.hs` file with the following in the same directory as the rest of our files (to keep it simple):

```haskell
module Main where

import Hello

main =
  sayHello "GoatScreams McGee"
```

Now we need to create the sandbox that we'll install our dependencies (just `hello-haskell` in this case):

```
$ cabal sandbox init
Writing a default package environment file to
call-em-up/cabal.sandbox.config
Creating a new sandbox at call-em-up/.cabal-sandbox
```

Just so we can see what it looks like when we try to build our project without `hello-haskell` installed, here's the error I get if I invoke `cabal build` or `cabal repl`:

```
Package has never been configured.
Configuring with default flags. If this
fails, please run configure manually.

Warning: The package list for 'hackage.haskell.org'
does not exist. Run 'cabal update' to download it.
Resolving dependencies...
Configuring call-em-up-0.1.0.0...
cabal: At least the following dependencies are missing:
hello-haskell -any
```

We had no version constraints on `hello-haskell`, so the constraint is `-any`. We can't build the project or load it in the Cabal REPL because

hello-haskell isn't installed. The next problem is making Cabal aware of *where* our project **hello-haskell** is located. Because it doesn't yet know where to look, here's the error you'll get if you update your index of what packages Hackage, the community package database, offers and try to install **hello-haskell**:

```
$ cabal update
Downloading the latest package list
from hackage.haskell.org

$ cabal install hello-haskell
cabal: There is no package named 'hello-haskell'.

You may need to run 'cabal update' to get the
latest list of available packages.
```

It suggests running **cabal update** because ordinarily your packages aren't local or private; they're coming from Hackage. **cabal update** just makes Cabal aware of the latest packages and versions that Hackage offers. However, **hello-haskell** isn't on Hackage. We'd already run **cabal update**, so we know it's not on Hackage. What we must do is add the local version of **hello-haskell** as a source of dependencies for the sandbox. Try running **cabal help sandbox** to see the help for the sandbox configuration to see the documentation for the following command:

```
$ pwd
/home/callen/Work/call-em-up
$ cabal sandbox add-source ../hello-haskell
$ cabal install --only-dependencies
Resolving dependencies...
Notice: installing into a sandbox located at
/home/callen/Work/call-em-up/.cabal-sandbox
Configuring hello-haskell-0.1.0.0...
Building hello-haskell-0.1.0.0...
Installed hello-haskell-0.1.0.0
```

Now we've successfully satisfied our dependencies (just our project `hello-haskell`) and can run `cabal repl` to interact with our code:

```
$ cabal repl
Preprocessing executable 'call-em-up'
for call-em-up-0.1.0.0...

GHCi, version 7.10.1: http://www.haskell.org/ghc/
[1 of 1] Compiling Main
Ok, modules loaded: Main.
Prelude> main
Hi GoatScreams McGee!
```

Let's reproduce an error that is often made and can confuse people new to Haskell and Cabal. We're going to revisit our `hello-haskell.cabal` file in our `hello-haskell` project and make a change to a single line, commenting out the exposed modules:

```
-- exposed-modules:     Hello
```

If we run `cabal install hello-haskell`, it'll auto-detect that our `hello-haskell` project changed and reinstall it:

```
$ pwd
/home/callen/Work/call-em-up
$ cabal install hello-haskell
Some add-source dependencies have been
modified. They will be reinstalled...

Resolving dependencies...
In order, the following will be installed:
hello-haskell-0.1.0.0 (reinstall)
Warning: Note that reinstalls are always dangerous.
Continuing anyway...
Notice: installing into a sandbox located at
/home/callen/Work/call-em-up/.cabal-sandbox
```

```
Configuring hello-haskell-0.1.0.0...
Building hello-haskell-0.1.0.0...
Installed hello-haskell-0.1.0.0


$ pwd
/home/callen/Work/call-em-up
$ cabal repl
Preprocessing executable 'call-em-up' for
call-em-up-0.1.0.0...
GHCi, version 7.10.1: http://www.haskell.org/ghc/
<command line>: cannot satisfy -package-id ...
    (use -v for more information)
```

You'll see **cabal** say that it wants a package-id with a specific hash attached
to the package name like f5a4cf0c1a1bfdab80345d849b8afcbd. To reset this,
invoke **cabal clean** and then try again:

```
$ pwd
/home/callen/Work/call-em-up
$ cabal clean
cleaning...
$ cabal repl
Package has never been configured.
Configuring with default flags. If this
fails, please run configure manually.

Resolving dependencies...
Configuring call-em-up-0.1.0.0...
Preprocessing executable 'call-em-up'
for call-em-up-0.1.0.0...

GHCi, version 7.10.1: http://www.haskell.org/ghc/

Main.hs:3:8:
    Could not find module 'Hello'
    Use -v to see a list of the files searched for.
Failed, modules loaded: none.
```

```
Prelude>
```

We got the error about not being able to find the module `Hello` because it's not exposed in our `hello-haskell` project!

13.9   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 aren't strictly speaking necessary. They can make blocks of code more readable and also hide the underlying algebra, 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).

We've also noted before that the `main` executable in a Haskell program should always have the type `IO a`, where *a* is some type, often unit `()`. The **do** syntax specifically allows us to sequence *monadic actions*. `Monad` is a typeclass 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:

```
main = do   x1   <-   getLine
--    [1]  [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 of 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 obtained 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 monad:

```
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 a`, the final value must also have an `IO a` 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. Now, 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:

```haskell
main :: IO Bool
main =  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`:

```haskell
main :: IO Bool
main =  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 make an `if-then-else` within our `do` block:

```haskell
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 is works in a similar way for monads other than IO.

Do notation considered harmful!   Just kidding. But sometimes enthusisatic 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** (actually, 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, we could have put **do** in front of both **putStrLn** and **return** and it would have worked the same, but things get messy and the Haskell ninjas will come and be severely disappointed in you.

13.10   Hangman game

Now we're ready to build a game. Make a directory for your game, change so you're working within that directory, and let's fire up the Cabal.

For the **cabal init** we did:

```
$ cabal init -n -l BSD3 --is-executable \
--language=Haskell2010 -u 'Chris N Julie' \
-c Game -s 'Playing Hangman' -p hangman
```

You'll need a words file for getting words from. Most Unix-based operating systems will have a words list located at **/usr/share/dict/words**. You may have one that is differently located, or you may need to download one. We put it in the working directory at **data/dict.txt**:

```
├── data
│   └── dict.txt
├── hangman.cabal
├── LICENSE
├── Setup.hs
└── src
    └── Main.hs
```

You haven't created that `src` branch yet – soon.

The file was newline separated and so looked like:

```
$ head data/dict.txt
A
a
aa
aal
aalii
aam
Aani
aardvark
aardwolf
Aaron
```

Now 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 == 1.1
                   , split == 0.2.2
  default-language:  Haskell2010
```

The important bit here is that we used two libraries: `random` and `split`. Normally you'd do version ranges for your dependencies like you see with `base`, but we assigned specific versions of `random` and `split` for the sake of explicitness. The primary and only source file was in `src/Main.hs`.

13.11 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.Random (randomRIO) -- [6]
```

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 libraries 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 have 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 just evaluating once and then stopping.

2. We will use `toLower` from `Data.Char` to convert all 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 if 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. Here `all` is a function which 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> all isJust [Just 'd', Nothing, Just 'g']
False
Prelude> all isJust [Just 'd', Just 'o', Just 'g']
True
```

The function `all` has the type:

Foldable t **=>** (a **->** **Bool**) **->** t a **->** **Bool**

We haven't explained the `Foldable` typeclass. 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 like so:

```
Prelude> :t all :: (a -> Bool) -> [a] -> Bool
all :: (a -> Bool) -> [a] -> Bool
```

This will work for any type which has a `Foldable` instance:

```
Prelude> :t all :: (a -> Bool) -> Maybe a -> Bool
all :: (a -> Bool) -> Maybe a -> Bool

-- note the type variables used and
-- experiment independently
Prelude> :t all :: (a -> Bool) -> Either b a -> Bool
all :: (a -> Bool) -> Either b a -> Bool
```

But it will not work if the datatype doesn't have an instance of `Foldable`:

```
Prelude> :t all :: (a -> Bool) -> (b -> a) -> Bool

No instance for (Foldable ((->) b1)) arising
from a use of 'all'
```

```
In the expression:
  all :: (a -> Bool) -> (b -> a) -> Bool
```

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 the Recursion chapter to put hyphens in our Numbers Into Words
   exercise:

   ```
   Prelude> import Data.List (intersperse)
   Prelude> intersperse ' ' "Blah"
   "B l a h"
   ```

   Conveniently, intersperse's type 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 just report that an
   exception occurred. In a normal running program that doesn't catch
   the exception, it'll end your whole program.

6. 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
```

We will later use this random number generation to produce a random index of a word list to provide a means of selecting random words for our puzzle.

13.12   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 our 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> length $ lines "aardvark\naaron\nwoot"
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 defined above. That will give us a shorter list
of words to use in the puzzles:

```haskell
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 mentioned 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> let 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 had created above. Rougly 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:

```haskell
randomWord :: WordList -> IO String
randomWord wl = do
  randomIndex <- randomRIO ( , )
--                             ^^^ you need to 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 Monads chapter. For now, we can say that, as we said about `do` syntax, `bind` allows us to sequentially compose actions such that a value generated by the first becomes an argument to the second:

```haskell
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.

13.13 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`:

```haskell
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 typeclass `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've named our list of `Char`, but we've done nothing with the `String` itself:

```haskell
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 of 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:

```haskell
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 3 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 put the names of those in 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 very similar to the one you just wrote, but this time we don't care if 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 fmapped 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 typeclass 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:

```
*Main> renderPuzzleChar Nothing
```

```
'_'
*Main> renderPuzzleChar (Just 'c')
'c'
*Main> let n = Nothing
*Main> let daturr = [n, Just 'h', n, Just 'e', n]
*Main> fmap renderPuzzleChar daturr
"_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 filledInSoFar s) c =
--                      [1]                    [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 the `filledInSoFar` replaced by `newFilledInSoFar` the *c* consed onto the front of the *s* list.

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 guessed, then we go ahead and return `Just wordChar` to fill in that spot in the puzzle. Otherwise, we just kick the guessChar back out. We kick guessChar back out because it might either be a previously correctly guessed character *or* a `Nothing` that have neither guessed correctly this time nor in the past.

5. `guessed` is the character they guessed.

6. `wordChar` is the characters in the puzzle word – not the ones they've guessed or not guessed but the characters in the word that they're supposed to be guessing.

7. `guessChar` is the list that keeps track of the characters the player has guessed so far.

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. Otherwise, it is a guessed character but does not belong in the word.

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 the Lists chapter. It's going to zip

the `word` with the `filledInSoFar` values while applying the `zipper` function from just above it to the values as it does.

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, it tells the player what you 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;

- or, 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:

```haskell
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 just 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:

```haskell
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 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 to fill into 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 that.

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:

```haskell
gameWin :: Puzzle -> IO ()
gameWin (Puzzle _ filledInSoFar _) =
  if all isJust filledInSoFar then
    do putStrLn "You win!"
       exitSuccess
  else return ()
```

Next is the instruction for running a game. Here we use `forever` so that this will execute this series of actions indefinitely:

```haskell
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, here is our main function that brings everything together: it gets a word from the word list we generated, generates a fresh puzzle, and then executes the `runGame` actions we just saw above, until such time as you guess all the characters in the word correctly or have made 7 guesses, whichever comes first:

```haskell
main :: IO ()
main = do
  word <- randomWord'
  let puzzle = freshPuzzle (fmap toLower word)
  runGame puzzle
```

13.14   Adding a newtype

Another way you could modify your code in the above and gain, perhaps, more clarity in places is with the use of `newtype`:

```haskell
-- replace this synonym with the newtype
-- type WordList = [String]

newtype WordList =
  WordList [String]
  deriving (Eq, Show)
```

```haskell
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
```

13.15 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 9 characters long, you only get to guess 7 characters;

- it ends the game after 7 guesses, whether they were correct or incorrect;

- if your 7th guess supplies the last letter in the word, it may still tell you 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 towards the guess limit.

Modifying code

1. Ciphers: Open your Ciphers module and modify it so that the Caesar and Vignere 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.

```haskell
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 tried using `palindrome` on a sentence such as "Madam I'm Adam," you may have noticed that palindrome checker doesn't work on that. 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.
```haskell
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.

```haskell
gimmePerson :: IO ()
gimmePerson = undefined
```

Since `IO ()` is about the least informative type imaginable, we'll tell 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 entered. You'll need the `read` function for Age because it's an Integer rather than a String.

c) If it constructed a successful person, it should print "Yay! Successfully got a person:" followed by the Person value.

d) If it got an error value, report that an error occurred and print the error.

13.16 Follow-up resources

1. Cabal FAQ
   `https://www.haskell.org/cabal/FAQ.html`

2. Cabal user's guide
   `https://www.haskell.org/cabal/users-guide/`

3. How I Start: Haskell
   `http://bitemyapp.com/posts/2014-11-18-how-i-start-haskell.html`

4. Stack, for help with managing dependencies for projects
   `https://github.com/commercialhaskell/stack`

5. A Gentle Introduction to Haskell, Modules chapter.
   `https://www.haskell.org/tutorial/modules.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 for Haskell (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 nonprogrammers 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 spec 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 together properly.

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 – which will then be 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. 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 minBounds 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 minimum 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 simple 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 covered later as independent concepts.

First, let's come up with a simple test case for addition. Generally we want to make a Cabal project, 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 Cabal project:

```
-- 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
```

---

[1]http://hackage.haskell.org/package/hspec

```
exposed-modules:    Addition
ghc-options:        -Wall -fwarn-tabs
build-depends:      base >=4.7 && <5
                  , hspec
hs-source-dirs:     .
default-language:   Haskell2010
```

Note 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 can probably get away with not specifying it for now.

Next we'll make the `Addition` module (exposed-modules) in the same directory as our Cabal file. This is why the `hs-source-dirs` option in the library stanza was set to `.` – this is the convention for referring to the current directory.

For now, we'll write a simple placeholder function, just to make sure everything's working:

```
-- Addition.hs
module Addition where

sayHello :: IO ()
sayHello = putStrLn "hello!"
```

Your local project directory should look like this now, before having run any Cabal commands:

```
$ tree
.
├── Addition.hs
└── addition.cabal
```

The next steps are to initialize a sandbox for storing our project-specific packages, which in this case will be Hspec and its dependencies:

```
$ cabal sandbox init
```

Then we'll want to install the dependencies for our project:

```
$ cabal install --dep
```

If that succeeded, let's fire up a REEEEEEEPL and see if we can call
sayHello:

```
$ cabal repl

[some noise about configuring, loading packages, etc.]
1 of 1] Compiling Addition

Ok, modules loaded: Addition.
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 module. You may have
encountered an error or a mistake might've been made. 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:7:1: parse error on input 'import'
Failed, modules loaded: none.
```

What else may have gone wrong? Well, we might have the package **hspec** installed, but not included in our **build-depends** for our project. Note you'll need to quit and reopen the REPL if you've made any changes to your **.cabal** file to reproduce this error or fixed a mistake:

```
$ cabal repl
./addition.cabal has been changed. Re-configuring
with most recently used options. If this fails,
please run configure manually.

Resolving dependencies...
...

Addition.hs:3:8:
Could not find module 'Test.Hspec'
It is a member of the hidden package 'hspec-2.1.10'.
Perhaps you need to add 'hspec' to the build-depends
 in your .cabal file.
 Use -v to see a list of the files searched for.
```

```
Failed, modules loaded: none.
```

Or the library may not be installed at all:

```
$ cabal sandbox hc-pkg -- unregister hspec

$ cabal repl
...

Addition.hs:3:8:
    Could not find module 'Test.Hspec'
    Use -v to see a list of the files searched for.
Failed, modules loaded: none.
Prelude>
```

If you changed 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, **cabal install --dep** 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 ..)
Prelude>
```

: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 the 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 at their 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:

```haskell
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`.

Here, we're nesting multiple `do` blocks. The type 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 is up to you. With the above code in place, we can load or reload our module and run `main` to see the test results:

---

[2]http://hspec.github.io/

```
Prelude> main

Addition
  1 + 1 is greater than 1

Finished in 0.0041 seconds
1 example, 0 failures
```

OK, so what happened here? Basically, Hspec runs your code and verifies that the arguments you passed to `shouldBe` are equal. Let's look at the types:

```
shouldBe :: (Eq a, Show a) => a -> a -> Expectation

-- contrast 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:

```
    it "2 + 2 is equal to 4" $ do
      2 + 2 `shouldBe` 4
```

Add the above to your describe block about `Addition` 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 the Recursion chapter, we wrote our own division function that looked like this:

```haskell
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 that to see that it works as it should. To keep things simple, we added `dividedBy` to our `Addition.hs` file and then rewrote 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:

```haskell
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 Cabal 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 Recursion, 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 demonsrate 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]

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 is 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.

---

[3]`https://github.com/bitemyapp/bloodhound`

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 (`cabal install --dep`). Then open a new `cabal repl` 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 had not asserted the type of x in the property test, the compiler would not have known what concrete type to use, and we'd see a message like this:

```
No instance for (Show a0) arising from a use of 'property'
The type variable 'a0' is ambiguous
...
No instance for (Num a0) arising from a use of '+'
The type variable 'a0' is ambiguous
...
No instance for (Ord a0) arising from a use of '>'
The type variable 'a0' is ambiguous
```

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 said you expected. 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 typeclass 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 merely 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 typeclass instance, as types and typeclasses 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 one just 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 pseudo-random 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 our 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 a great deal of sense at this point, it will be more clear once we've covered monads, and even more so once we cover `IO`.

We use the `Arbitrary` typeclass in order to provide a generator for `sample`. It isn't a terribly principled typeclass, 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 just 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` typeclass.

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 trival function that always returns a 1 of type `Int`:

```haskell
-- the 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 our value isn't **arbitrary** for some type, but the **trivialInt** value we defined above. That generator always returns 1, so all **sample'** can return for us is a list of 1.

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 gave 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 wouldn'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:

```
-- choose :: System.Random.Random a => (a, a) -> Gen a
-- elements :: [a] -> Gen a

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 just play with getting lists of sample data for each.

Our next examples are a bit more complex:

```haskell
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 unit for you. Outside of GHCi, you'll get an error about an ambiguous type – we covered some of this when we explained typeclasses 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> sample (genTuple :: Gen (Int, Float))
(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` typeclass:

```
Prelude> sample (genTuple :: Gen ([()], Char))
([],'\STX')
([()],'X')
([],'?')
([],'\137')
([(),()],'\DC1')
([(),()],'z')
```

You can use `:info Arbitrary` in your GHCi to see what 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 actually 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 just play with this in the REPL; it will become useful to know 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 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` function we were calling with Hspec, we'll call `runQc`, which will call on QuickCheck to test the property we defined. When we run QuickCheck directly, it reports how many tests it ran:

```
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, modules loaded: Addition.
Prelude> runQc
*** Failed! Falsifiable (after 1 test):
0
```

Conveniently, QuickCheck doesn't just tell us that our test failed, but it tells us the first input it encountered that it failed 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 etc).

14.5 Morse code

In the interest of playing with testing, we'll work through an example project where we translate text to and from Morse code. We're going to start a new Cabal project for this:

```
cabal init
```

When you do a `cabal init` instead of only initializing a sandbox, it automatically generates a file called `Setup.hs` that looks like this:

```
-- Setup.hs

-- this gets generated by Cabal init

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 automatically generated by your `cabal init`, but you'll have to add to what it generates, being careful about things like capitalization and indentation:

```
-- morse.cabal

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
```

Now that's set up and ready for us, so 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 another module (and, by exclusion, what will not) as a way of documenting your intent. In this case, we have exported here more than we imported into `Main`, as we realized that we only needed the two specified functions for `Main`. 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 came 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, must have an `Ord` instance), just as our binary tree functions earlier, such as `insert`, needed 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 (equal).

You can see below why we used 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:

```haskell
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 used `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 morse 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:

```haskell
morseToLetter :: M.Map Morse Char
morseToLetter = M.foldWithKey (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 used `Maybe` in three of those: 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 actually 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 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 to Morse:

```
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:

```haskell
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** function:

```haskell
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 this, go ahead and try this:

```
$ echo "hi" | ./dist/build/morse/morse to
.... ..

$ echo ".... .." | ./dist/build/morse/morse from
hi
```

Or if you wanted, you could use `cabal run` to run the program:

```
$ echo ".... . .-.. .-.. ---" | cabal run from
hello
```

Otherwise, just 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 file name (the
name per se isn't important, but it must agree with the test file you have
named in your cabal configuration for this project):

```haskell
-- 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:

```haskell
allowedChars :: [Char]
allowedChars = M.keys letterToMorse

allowedMorse :: [String]
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.

Now 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 all this done:

```
$ tree
.
├── LICENSE
├── Setup.hs
├── morse.cabal
├── src
│   ├── Main.hs
│   └── Morse.hs
└── 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 `cabal repl` from our main project directory:

```
/Morse$ cabal repl tests
cabal: Cannot build the test suite 'tests' because test suites are not
enabled. Run configure with the flag --enable-tests
```

Woops – maybe Cabal isn't ready to run our tests. Fortunately it tells us exactly how to fix this problem:

```
/Morse$ cabal configure --enable-tests
Warning: The package list for 'hackage.haskell.org' is 16 days old.
Run 'cabal update' to get the latest list of available packages.
Resolving dependencies...
Configuring Morse-0.1.0.0...
```

Let's try it again:

```
/Morse$ cabal repl tests
Preprocessing library Morse-0.1.0.0...
In-place registering Morse-0.1.0.0...
Preprocessing test suite 'tests' for Morse-0.1.0.0...
GHCi, version 7.8.3: http://www.haskell.org/ghc/   :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Loading package array-0.5.0.0 ... linking ... done.
Loading package deepseq-1.3.0.2 ... linking ... done.
Loading package containers-0.5.5.1 ... linking ... done.
Loading package old-locale-1.0.0.6 ... linking ... done.
Loading package time-1.4.2 ... linking ... done.
Loading package random-1.1 ... linking ... done.
Loading package pretty-1.1.1.1 ... linking ... done.
Loading package template-haskell ... linking ... done.
Loading package transformers-0.4.3.0 ... linking ... done.
Loading package primitive-0.6 ... linking ... done.
Loading package tf-random-0.5 ... linking ... done.
Loading package QuickCheck-2.8.1 ... linking ... done.
Loading package Morse-0.1.0.0 ... linking ... done.
```

```
[1 of 1] Compiling Main             ( tests/tests.hs, interpreted )
Ok, modules loaded: Main.
```

Sweet. This time it seems like everything loaded for us. Let's see what happens:

```
*Main> main
+++ OK, passed 100 tests.
```

The test generates 100 random Morse code conversions (a bunch of random strings) 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 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 Recursion? You'll be writing tests to validate the functions you wrote.

```haskell
module WordNumberTest where

import Test.Hspec
import WordNumber (digitToWord, digits, wordNumber)

main :: IO ()
main = hspec $ do
  describe "digitToWord does what we want" $ do
    it "returns zero for 0" $ do
      digitToWord 0 `shouldBe` "zero"
    it "returns one for 1" $ do
      print "???"

  describe "digits does what we want" $ do
    it "returns [1] for 1" $ do
      digits 1 `shouldBe` [1]
    it "returns [1, 0, 0] for 100" $ do
      print "???"

  describe "wordNumber does what we want" $ do
    it "returns one-zero-zero for 100" $ do
      wordNumber 100 `shouldBe` "one-zero-zero"
    it "returns 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 simple arithmetic 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 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
   ```

4. Now do the same for multiplication.

5. We mentioned in one of the first chapters that there are some laws involving the relationship of `quot` and `rem` and `div` and `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 (^) 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 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. Hm. Is that so?

    ```
    f n xs = length (take n xs) == n
    ```

11. Finally, this is 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 in 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. It's already sorted, so new applications of the sort function won't change it.

Use QuickCheck and the following helper functions to demonstrate idempotence for the following:

```
twice f = f . f
fourTimes = twice . twice
```

1. ```
   f x =
       once x
       == twice capitalizeWord x
       == fourTimes capitalizeWord x
   ```

2. ```
   f x =
       sort x
       == twice sort x
       == fourTimes sort x
   ```

Make a Gen random generator for the datatype

We demonstrated in the 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

As a final exercise, you should now 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 pure 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 =
          zipWith (zipper c) word filledInSoFar
```

and:

```haskell
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.

Validating ciphers

Create QuickCheck properties that verify your Caesar and Vignere ciphers
return the same data after encoding and decoding a string.

14.7 Definitions

1. *Unit testing* is a method in where 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 input and 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 multipled by one or zero.

14.8 Follow-up resources

1. Pedro Vasconcelos; An introduction to QuickCheck testing;
   `https://www.fpcomplete.com/user/pbv/`
   `an-introduction-to-quickcheck-testing`

2. Koen Claessen and John Hughes; (2000)
   QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs

3. Pedro Vasconcelos; Verifying a Simple Compiler Using Property-based Random Testing;
   `http://www.dcc.fc.up.pt/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 which have well-defined, lawful representations in mathematics. A word frequently used to describe these abstractions is *algebra*, where we mean one or more *operations* and the *set* they operate over. By set, we mean the type 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 are 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 talk about when we talk about algebras

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 about 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 typeclasses; the typeclasses define 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 in Haskell 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 alone 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]        [2]        [3]       [4]                [5]
```

1. The thing we're talking about — monoids. That'll end up being the name of our typeclass.

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 or rules:

```haskell
-- [] = mempty, or the identity
-- mappend is the binary operation
-- to append, or join, two arguments
mappend [1..5] [] = [1..5]
mappend [] [1..5] = [1..5]

-- or, more generally

mappend x mempty = x
mappend mempty x = x
```

So, 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 reparenthesized, or reassociated) in different orders and give the same result, as in addition. Identity means there exists some value such that when we pass it as 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 typeclass that generalizes these laws across types.

15.4 How Monoid is defined in Haskell

Typeclasses 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` typeclass 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 typeclass merely abstracts that pattern out and makes it general and available to other datatypes that don't have predefined functions that do this for you (e.g., (++)), including your own datatypes when appropriate.

In Haskell, we think of types as *having* an instance of a typeclass. When we represent abstract operations that can be reused across a set of types, we usually represent them as a typeclass.

The typeclass `Monoid` is defined:

```
class Monoid m where
  mempty  :: m
  mappend :: m -> m -> m
  mconcat :: [m] -> m
  mconcat = foldr mappend mempty
```

`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 Monoid, really, is that they are quite 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 typeclass just abstracts the *patterm* out, giving you the ability to use the operations over a larger range of types.

List

One of the more pervasive types with an instance of `Monoid` is `List`. 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> mappend "Trout" " goes well with garlic"
"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> (++) "Trout" " goes well with garlic"
"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 the definition of Monoid for lists, we can see how this all lines up:

```
instance Monoid [a] where
  mempty  = []
  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 not a 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 typeclass, not two (one instance for a sum, one for a product).

This won't work:

```
Prelude> mappend 1 1

<interactive>:7:1:
    No instance for (Num a0) arising from a use of 'it'

    ...and a bunch of noise.
```

Because it isn't clear if those should be added or multiplied as a `mappend` operation. It says there's no `Monoid` for `Num a => a` 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 `Monoid` library. While there are two possible instances of `Monoid` for numeric values, we avoid using scoping tricks and abide by the rule that typeclass 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 (modulo `bottom`, explained later) between the following datatypes:

```haskell
data Server = Server String
```

```haskell
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 as is necessary for typical products and sums.

For veteran programmers that 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. 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. Improve type safety: avoid mixing up many values of the same representation, such as `Text` or `Integer`.

3. Add different typeclass instances to a type that is otherwise unchanged representationally.

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` data constructors:

```
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 Control.Applicative Data.Monoid> :t (<>)
(<>) :: Monoid m => m -> m -> m
```

```
Prelude> Sum "Frank" <> Sum " " <> Sum "Herbert"
```

```
No instance for (Num [Char]) arising from a use of '<>'
```

The example didn't work because the `a` in `Sum a` was `String` which is not an instance of `Num`.

`Sum` and `Product` do what you'd expect with just a bit of syntactic surprise:

```
Prelude Data.Monoid> mappend (Sum 8) (Sum 9)
Sum {getSum = 17}
```

```
Prelude Data.Monoid> mappend mempty Sum 9
Sum {getSum = 9}
```

But `mappend` only joins two things, so you can't do this:

```
Prelude Data.Monoid> mappend (Sum 8) (Sum 9) (Sum 10)
```

You'll get a big error message including this line:

```
Possible cause: 'Sum' is applied to too many arguments
    In the first argument of 'mappend', namely '(Sum 8)'
```

So, that's easy enough to fix by nesting:

```
Prelude> mappend (Sum 1) (mappend (Sum 2) (Sum 3))
Sum {getSum = 6}
```

Or somewhat less tedious by infixing the mappend function:

```
Prelude> (Sum 1) <> (Sum 1) <> (Sum 1)
Sum {getSum = 3}
Prelude> (Sum 1) `mappend` (Sum 1) `mappend` (Sum 1)
Sum {getSum = 3}
```

Or you could also put your `Sum`s in a list and use `mconcat`:

```
Prelude Data.Monoid> 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 the value:

```
Prelude> getSum $ mappend (Sum 1) (Sum 1)
2
Prelude> getProduct $ mappend (Product 5) (Product 5)
25
Prelude> getSum $ mconcat [(Sum 5), (Sum 6), (Sum 7)]
18
```

`Product` is similar to `Sum` but for multiplication.

15.7   Why bother?

Because monoids are really 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've encountered 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 even stronger 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 the Foldable chapter, but here's a taste:

```
Prelude> foldr mappend mempty ([2, 4, 6] :: [Product Int])
Product {getProduct = 48}

Prelude> foldr mappend mempty ([2, 4, 6] :: [Sum Int])
Sum {getSum = 12}

Prelude> foldr mappend mempty ["blah", "woot"]
"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 must adhere to because mathematicians care about the laws. That matters because mathematicians often want the same things programmers want! A proof that is inelegant, a proof term that doesn't compose well, or that cannot be understood is not very good or useful to a mathematician. 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 (many) other things, laws provide us guarantees that let us build on solid foundations for the predictable composition (or combination) of programs. Without the ability to combine programs, everything must be hand-written from scratch, nothing can 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 just associativity or an identity, but these are simple examples we are starting with for now, partly because `Monoid` is a good place to start with algebras-as-typeclasses. 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
```

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 more easily if we first introduce the infix operator for `mappend`, `<>`. Note the parenthesization on the two examples:

```
Prelude> :t (<>)
(<>) :: Monoid 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> mconcat [Sum 1, Sum 2, Sum 3]
Sum {getSum = 6}

Prelude> foldr mappend mempty [Sum 1, Sum 2, Sum 3]
Sum {getSum = 6}
```

Now let's see all of that again, but using the Monoid of lists:

```
-- mempty is []
-- 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 *what Monoid* you'll be working with.

15.9 Different typeclass instance, same representation

Monoid is somewhat different from other typeclasses 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 tell them apart, as we did with Sum and Product.

Addition is a classic appending operation, as is list concatenation. Referring to multiplication as a mappend 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 catamorophisms more generally. Mappending 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 one of disjunction. As we do with numbers, we use newtypes to distinguish the two insances. `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 some sense in which it might feel strange to think of this as a combining or *mappending* operation, unless we recall that mappending is less about combining and more about condensing or reducing.

The Maybe type actually 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 occurred. 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> First (Just 1) `mappend` First (Just 2)
First {getFirst = Just 1}
```

Last returns the "last" or rightmost non-Nothing value:

```
Prelude> Last (Just 1) `mappend` Last (Just 2)
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 Nothing `mappend` Last (Just 2)
```

```
Last {getLast = Just 2}

Prelude> First Nothing `mappend` First (Just 2)
First {getFirst = Just 2}
```

Neither can, for obvious reasons, return anything if all values are Nothing:

```
Prelude> First Nothing `mappend` First Nothing
First {getFirst = Nothing}

Prelude> Last Nothing `mappend` Last Nothing
Last {getLast = Nothing}
```

To maintain the unique pairing of type and typeclass instance, newtypes are used for all of those, just 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 `Monoid`s for `Maybe` than just `First` and `Last`. We will now 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. Monoids 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 of them.

If you do have a datatype that has a type argument that does not appear anywhere in the terms, the typechecker does not demand that you have a Monoid instance for that argument. For example:

```haskell
data Booly a =
    False'
  | True'
  deriving (Eq, Show)

-- conjunction; just cause.
instance Monoid (Booly a) where
  mappend False' _ = False'
  mappend _ False' = False'
  mappend True' True' = True'
```

We didn't need a Monoid constraint for *a* because we're never mappending *a* values (we can't; 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

Write the Monoid instance for our Maybe type renamed to Optional.

```haskell
data Optional a =
    Nada
  | Only a
  deriving (Eq, Show)

instance Monoid a => Monoid (Optional a) where
  mempty = undefined
  mappend = undefined
```

Expected output:

```
Prelude> Only (Sum 1) `mappend` Only (Sum 1)
Only (Sum {getSum = 2})

Prelude> Only (Sum 1) `mappend` Nada
Only (Sum {getSum = 1})

Prelude> Nada `mappend` Only (Sum 1)
Only (Sum {getSum = 1})
```

Associativity

This will be mostly review, but we want to be specific about associativity. Associativity simply says that you can associate the arguments of your operation differently and the result will be the same.

Let's review examples of some operations that can be *reassociated*:

```
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 reassociated 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, not just reassociate the parentheses, and still get the same result. Addition and multiplication are commutative, but **(++)** for the list type is not.

Let's demonstrate this by writing a mildly evil version of addition that flips the order of its arguments:

```
Prelude> let 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> let evilPlusPlus = flip (++)
Prelude> let oneList = [1..3]
Prelude> let otherList = [4..6]

Prelude> oneList ++ otherList
[1,2,3,4,5,6]

Prelude> oneList `evilPlusPlus` otherList
[4,5,6,1,2,3]
```

In this case, 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.

Commutativity is a strong property and can be useful 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 very 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 1 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> let myList = [1..424242]
Prelude> map (+0) myList == myList
True

Prelude> map (*1) myList == myList
True
```

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 typeclasses 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*. Do not be lazy about this! If you get an orphan instance warning from GHC, *fix it*.

An orphan instance is when an instance is defined for a datatype and typeclass, but not in the same module as either the declaration of the typeclass or the datatype. If you don't "own" the typeclass 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 notionally two Monoids for numbers without resorting to orphans or messing up typeclass 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:

```haskell
module Listy where

newtype Listy a =
  Listy [a]
  deriving (Eq, Show)


module ListyInstances where

import Data.Monoid
import Listy

instance Monoid (Listy a) where
  mempty = Listy []
  mappend (Listy l) (Listy l') = Listy $ mappend l l'
```

So our directory will look like:

```
$ tree
.
├── Listy.hs
└── ListyInstances.h
```

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 dis-
coverable. 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 `cabal init`'ing
a project. For *anything* more complicated or long-lived than this, use a
dependency and build management tool like 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:
Duplicate instance declarations:
instance Monoid (Listy a) -- Defined at Listy.hs:7:10
instance Monoid (Listy a) -- Defined at ListyInstances.hs:5:10
```

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 typeclass methods will start behaving differently depending on what modules are imported, which breaks the fundamental assumptions and niceties of typeclasses.

There are a few solutions for addressing orphan instances:

1. You defined the type but not the typeclass? Put the instance in the same module as the type so that the type cannot be imported without its instances.

2. You defined the typeclass but not the type? Put the instance in the same module as the typeclass definition so that the typeclass cannot be imported without its instances.

3. Neither the type nor the typeclass are yours? Define your own new-type wrapping the original type and now you've got a type that "belongs" to you for which you can rightly define typeclass 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 typeclasses along with the nice reasoning properties that are associated

with them. A type *must* have a unique (singular) implementation of a typeclass 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[1] 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 an example from the Wikipedia article on Mad Libs:

```
"_____! he said _____  as he jumped into his car
 exclamation          adverb
____ and drove off with his _____  wife."
noun                           adjective
```

We can make this into a function, like the following:

---

[1] https://en.wikipedia.org/wiki/Mad_Libs

```haskell
import Data.Monoid

type Verb = String
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 this " <>
  adj <> " wife."
```

Now you're going to refactor this code a bit! Rewrite it using `mconcat`.

```haskell
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 arithemetic 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 thing 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
-- or infix

\ (<>) a b c ->
  a <> (b <> c) == (a <> b) <> c
```

Surprise! You can bind infix names for function arguments.

```haskell
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 quickcheck? The quickest and easiest way would probably look something like the following:

```haskell
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 associativity of functions:

```haskell
-- for brevity
Prelude> type S = String
Prelude> type B = Bool

Prelude> quickCheck (monoidAssoc :: S -> S -> S -> B)
+++ OK, passed 100 tests.
```

The `quickCheck` function uses the `Arbitrary` typeclass 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. We may not want to do this 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 which Arbitrary instance to get random values for testing from. You can use

`verboseCheck` to see what values were 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 the Testing chapter. GHCi has slightly more aggressive type-defaulting which can be handy in an interactive session when you just want to fire off some code and have your REPL pick a winner for the typeclasses it doesn't know how to dispatch. Compiled in a source file, GHC would've squawked about an ambiguous type.

Quickchecking left and right identity

Following on from what we did with associativity, left and right identity turn out to be even simpler to test:

```
monoidLeftIdentity :: (Eq m, Monoid m) => m -> Bool
monoidLeftIdentity a = (a <> mempty) == a

monoidRightIdentity :: (Eq m, Monoid m) => m -> Bool
monoidRightIdentity a = (mempty <> a) == a
```

Then running these properties against a Monoid:

```
Prelude> quickCheck (monoidLeftIdentity :: String -> Bool)
+++ OK, passed 100 tests.

Prelude> quickCheck (monoidRightIdentity :: 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. Here 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:

```haskell
-- associative, left identity, and right identity
-- properties have been elided. Add them to your copy of this.

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 Monoid Bull where
  mempty = Fools
  mappend _ _ = Fools

type BullMappend = Bull -> Bull -> Bull -> Bool

main :: IO ()
main = do
  quickCheck (monoidAssoc :: BullMappend)
  quickCheck (monoidLeftIdentity :: Bull -> Bool)
  quickCheck (monoidRightIdentity :: 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, lets line up the laws against what our mempty and mappend are.

```
-- how the instance is defined
mempty = Fools
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 it 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 even behaving like a zero as you're not even checking if either argument is `Fools` before returning `Fools`. It's an evil blackhole that just 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.

Intermission: Exercise

Write a Monoid instance for Maybe type which doesn't require a Monoid for the contents. Reuse the Monoid law QuickCheck properties and use them to validate the instance.

```haskell
newtype First' a =
  First' { getFirst' :: Optional a }
  deriving (Eq, Show)

instance Monoid (First' a) where
  mempty = undefined
  mappend = undefined

firstMappend :: First' a -> First' a -> First' a
firstMappend = mappend

type FirstMappend =
     First' String
  -> First' String
  -> First' String
  -> Bool

main :: IO ()
main = do
  quickCheck (monoidAssoc :: FirstMappend)
  quickCheck (monoidLeftIdentity :: First' String -> Bool)
  quickCheck (monoidRightIdentity :: First' String -> Bool)
```

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 like a disjunctive or "or"-oriented Monoid instance.

```
Prelude> First' (Only 1) `mappend` First' Nada
First' {getFirst' = Only 1}
Prelude> First' Nada `mappend` First' Nada
First' {getFirst' = Nada}
Prelude> First' Nada `mappend` First' (Only 2)
First' {getFirst' = Only 2}
Prelude> First' (Only 1) `mappend` First' (Only 2)
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 doesn't have an identity value. In that sense, it's a weaker algebra.

Not yet part of base   As of the writing of this book, the `Semigroup` typeclass isn't yet part of base. To follow along with the examples, you'll want to install the semigroups[2] library and use it from a `cabal repl`.

If you do install Semigroups and import the modules, keep in mind that it defines its own more general version of `(<>)` which only requires a Semigroup constraint rather than a Monoid constraint. If you don't know what to do, close your REPL and reopen it. After doing so, then import only the semigroups module and not Data.Monoid from base. If you must import Data.Monoid, hide the conflicting `(<>)` in your import declaration.

---

[2]`http://hackage.haskell.org/package/semigroups`

NonEmpty, a useful datatype

One really 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:

```
data NonEmpty a = a :| [a]
  deriving (Eq, Ord, Show)

-- some instances from the real library elided
```

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 just 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 ':!!:'
Failed, modules loaded: none.
```

On the other hand, an alphanumeric data constructor can't be used as an infix:

```haskell
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?
Failed, modules loaded: none.
```

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:

```haskell
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 from the semigroups library with the semigroup "mappend":

```haskell
-- you need to have `semigroups` installed
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 just like 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.

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 simply make all of our algebras as big as possible is that there are datatypes which 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 Monoid is *stronger* than Semigroup is to observe that it has a strict superset of the operations and laws that Semigroup provides. Anything which is a Monoid is by definition *also* a semigroup. Soon, Semigroup will be added to GHC's base library and when it is, Monoid will then superclass Semigroup.

```
class Semigroup a => Monoid a where
  ...
```

Earlier we reasoned about the inverse relationship between operations permitted over a type and the number of types that can satisfy. We can see this relationship between the number of operations and 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. Just referencing the value you were passed.

With `inc` a now has all the operations from Num, which lets us do more. But that also means it's 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: 7 methods in Num

In the next example we know it's an Integer, which gives us many more operations than just a Num instance:

```haskell
somethingInt :: Int -> Int
```

- Number of types: one — just `Int`.

- Number of operations: considerably more than 7. 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 that 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 meant 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.

15.14   Chapter exercises

Semigroup exercises

Given a datatype, implement the Semigroup instance. Add Semigroup constraints to type variables where needed. Use the Semigroup class from the semigroups library or write your own. When we use `<>`, we mean the infix mappend from the Semigroup typeclass.

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 Semigroup's only law is associativity, that's the only property you need to reuse.

```haskell
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 TrivialAssoc = Trivial -> Trivial -> Trivial -> Bool

main :: IO ()
main =
  quickCheck (semigroupAssoc :: TrivialAssoc)
```

2. `newtype Identity a = Identity a`

3. `data Two a b = Two a b`

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 Monoid for Or should have the following behavior. We can think of this as having a "sticky" Snd value where 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> let f = Combine $ \n -> Sum (n + 1)
Prelude> let 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*. If you can't figure out CoArbitrary, don't worry about QuickChecking this one.

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

12. ```
    -- Validation with a Semigroup
    -- that does something different

    newtype AccumulateRight a b =
      AccumulateRight (Validation a b)
      deriving (Eq, Show)

    instance Semigroup b =>
      Semigroup (AccumulateRight a b) where
        (<>) = undefined
    ```

13. ```
    -- Validation with a Semigroup
    -- that does something more

    newtype AccumulateBoth a b =
      AccumulateBoth (Validation a b)
      deriving (Eq, Show)

    instance (Semigroup a, Semigroup b) =>
      Semigroup (AccumulateBoth a b) where
        (<>) = undefined
    ```

Monoid exercises

Given a datatype, implement the Monoid instance. Add Monoid constraints to type variables where needed. For the datatypes you've already implemented Semigroup instances for, you just 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.

```haskell
data Trivial = Trivial deriving (Eq, Show)

instance Semigroup Trivial where
  (<>) = undefined

instance Monoid Trivial where
  mempty = undefined
  mappend = (<>)

type TrivialAssoc = Trivial -> Trivial -> Trivial -> Bool

main :: IO ()
main = do
  quickCheck (semigroupAssoc :: TrivialAssoc)
  quickCheck (monoidLeftIdentity :: Trivial -> Bool)
  quickCheck (monoidRightIdentity :: Trivial -> Bool)
```

2. ```haskell
   newtype Identity a = Identity a deriving Show
   ```

3. ```haskell
   data Two a b = Two a b deriving Show
   ```

4. ```haskell
   newtype BoolConj =
     BoolConj Bool
   ```

   What it should do:

   ```
   Prelude> (BoolConj True) `mappend` mempty
   BoolConj True
   Prelude> mempty `mappend` (BoolConj False)
   BoolConj False
   ```

5. ```haskell
   newtype BoolDisj =
     BoolDisj Bool
   ```

   What it should do:

   ```
   Prelude> (BoolDisj True) `mappend` mempty
   BoolDisj True
   Prelude> mempty `mappend` (BoolDisj False)
   BoolDisj False
   ```

6.
```haskell
data Or a b =
    Fst a
  | Snd b
```

What it should do:

```haskell
Prelude> Fst 1 `mappend` Snd 2
Snd 2
Prelude> Fst 1 `mappend` Fst 2
Fst 2
Prelude> Snd 1 `mappend` Fst 2
Snd 1
Prelude> Snd 1 `mappend` Snd 2
Snd 1
```

7.
```haskell
newtype Combine a b =
  Combine { unCombine :: (a -> b) }
```

What it should do:

```haskell
Prelude> let f = Combine $ \n -> Sum (n + 1)
Prelude> unCombine (mappend f mempty) $ 1
Sum {getSum = 2}
```

8. 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.

```haskell
newtype Comp a =
  Comp (a -> a)
```

9. These next one will involve doing something that will feel a bit unnatural still 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 needed.

10. **newtype** `Mem` s a =
```
      Mem {
        runMem :: s -> (a,s)
      }
```

```
instance Monoid a => Monoid (Mem s a) where
  mempty = undefined
  mappend = undefined
```

Given the following code:

```
f' = Mem $ \s -> ("hi", s + 1)
```

```
main = do
  print $ runMem (f' <> mempty) 0
  print $ runMem (mempty <> f') 0
  print $ (runMem mempty 0 :: (String, Int))
  print $ runMem (f' <> mempty) 0 == runMem f' 0
  print $ runMem (mempty <> f') 0 == runMem f' 0
```

A correct Monoid for Mem should, given the above code, get the following output:

```
Prelude> main
("hi",1)
("hi",1)
("",0)
True
True
```

Make certain your instance has output like the above, this is sanity-checking the Monoid identity laws for you! It's not a proof and it's not even as good as quick-checking, but it'll catch the most common mistakes people make. If you'd like to learn how to generate functions with QuickCheck, not just values, look at `CoArbitrary` in QuickCheck's documentation.

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 validate your instance as well, to do that, in particular you'll want to check the identity laws as a common mistake with this exercise is to write an instance that doesn't respect them.

15.15 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 it's 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 which 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 what mathematicians commonly mean by "algebra." This is sometimes diambiguated 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.

   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.

15.16 Follow-up resources

1. Algebraic structure; Simple English Wikipedia;
   https://simple.wikipedia.org/wiki/Algebraic_structure

2. Algebraic structure; English Wikipedia
   https://en.wikipedia.org/wiki/Algebraic_structure

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 typeclass. This chapter and the two that follow, on Applicative and Monad, will be on a similar topic. 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. Monads sort of steal 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 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.

The great logician Rudolf Carnap appears to have been the first person to use the word 'functor' in the 1930s. He invented the word to describe certain types of grammatical function words and logical operations over sentences or phrases. Functors are combinators: they take a sentence or phrase as input and produce a sentence or phrase as an output, with some logical operation applied to the whole. For example, negation is a functor in this sense because when negation is applied to a sentence, A, it produces the negated version, $\neg A$, as an output. It lifts the concept of *negation* over the entire sentence or phrase structure without changing the internal structure. (Yes, in English the negation word often appears inside the sentence, not on the outside, but he was a logician and unconcerned with how normal humans produced such pedestrian things as spoken sentences. In logic, the negation operator is typically written as a prefix, as above.)

This chapter will include:

- the return of the higher-kinded types;
- fmaps galore, and not just on lists;
- no more digressions about dusty logicians;

- words about typeclasses 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 fmapping over lists, as we did back in the Lists chapter. The function gets applied to each value inside the 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 typeclass `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 typeclass, 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 typeclass in Haskell looks like this:

```haskell
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Now let's dissect this a bit.

```haskell
class Functor f  where
[1]   [2]     [3]   [4]
  fmap :: (a -> b) -> f a -> f b
  [5]       [6]       [7]    [8]
```

1. `class` is the keyword to begin the definition of a typeclass

2. Functor is the name of the typeclass we are defining.

3. Typeclasses in Haskell usually refer to some sort of *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 typeclass definition.

4. The `where` keyword ends the declaration of the typeclass name and associated types. After the `where` the operations provided by the typeclass are listed.

5. We begin the declaration of an operation named `fmap`.

6. The argument `a -> b` is any function in Haskell.

7. 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` typeclass.

8. 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 typeclass 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 going 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]
```

List is, of course, one type that implements the typeclass Functor, but it seems unremarkable when it just does the same thing as `map`. However, lists aren't the only type that implement 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 a0'

Relevant bindings include it :: [b] (bound at 16:1)
In the second argument of 'map', namely '(Just 1)'
In the expression: map (+ 1) (Just 1)

Prelude> fmap (+1) (Just 1)
Just 2
```

Intriguing! What else?

```
--with a tuple!
Prelude> fmap (10/) (4, 5)
(4,2.0)

--with Either!
Prelude> fmap (++ ", Esq.") (Right "Chris Allen")
Right "Chris Allen, Esq."
```

We can see how the type of **fmap** specializes to different types here:

```
-- Functor f =>
fmap :: (a -> b) ->        f a ->          f b
     :: (a -> b) ->      [ ] a ->        [ ] b
     :: (a -> b) ->    Maybe a ->      Maybe b
     :: (a -> b) -> Either e a ->   Either e b
     :: (a -> b) ->      (e,) a ->       (e,) b
     :: (a -> b) -> Identity a ->   Identity b
     :: (a -> b) -> Constant e a -> Constant e 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 typeclass definition for `Functor` must be the same f throughout the entire definition, and it must refer to a type that implements the typeclass. This section details the practical ramifications of those facts.

The first thing we know is that our f here 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 be 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 (and inhabitable, *modulo* Void, etc.) type. Each argument must have the kind `*`.

2. The type f was 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 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
--has 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 typeclass, which we'll demonstrate with some meaningless typeclasses:

```
class Sumthin 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 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 -> (a -> c) -> (b -> d) -> e c d
--            [1]        [2]     [3]
```

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 here, just in another spot of the same function.

The kind checker is going to fail on the next couple of examples:

```haskell
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 typeclass 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:

```
'v' is applied to too many type arguments
In the type '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 '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.

Intermission: Exercises

Given a type signature, determine the kinds of each type variable:

1. What's the kind of a?

   ```haskell
   a -> a
   ```

2. What are the kinds of b and T? (The T is capitalized on purpose!)

   ```haskell
   a -> b a -> T (b a)
   ```

3. What's the kind of *c*?

```
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:

```haskell
-- 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 just one shining (kind) star! And if we load this file from GHCi, we'll get the following error:

```
Prelude> :l functors1.hs
[1 of 1] Compiling Main             ( functors1.hs, interpreted )

functors1.hs:8:18:
The first argument of 'Functor' should have kind '* -> *',
  but 'FixMePls' has kind '*'
In the instance declaration for 'Functor FixMePls'
Failed, modules loaded: none.
```

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:
(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 just:

```
($) :: (a -> b) -> a -> b
```

Without a type argument, this is mere function application.

Functor is function application

We just saw how trying to make a Functor instance for a type constant means you just 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 typeclass for function application "over", or "through", or "past" 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, modules loaded: Main.
```

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 conventionally 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 typeclass definition.

Now our code is happy-making!

```
Prelude> :l code/functors3.hs
[1 of 1] Compiling Main
Ok, modules loaded: Main.
Prelude> fmap (+1) (Pls 1)
Pls 2
```

Notice the function gets applied over and inside of the "structure." Now we can be stronk[1] Haskell coders and lift big heavy functions over abstract structure!

Okay, let's make another mistake for the sake of explicit-ness. What if we change the type of our Functor instance from `FixMePls` to `FixMePls a`?

---

[1]"Stronk" means *STRONK*.

```
-- functors4.hs

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 we didn't change the type; it still only takes one argument. But now that argument is part of the f structure. If we load this ill-conceived code:

```
Prelude> :l functors4.hs
[1 of 1] Compiling Main

functors4.hs:8:19:
The first argument of 'Functor' should have kind '* -> *',
  but 'FixMePls a' has kind '*'
In the instance declaration for 'Functor (FixMePls a)'
Failed, modules loaded: none.
```

We get the same error as earlier, because applying the type constructor gave us something of kind `*` from the original kind of `* -> *`.

Typeclasses and constructor classes

You may have initially paused on the type constructor f in the definition of Functor having kind `* -> *` — this is quite natural! In fact, earlier versions of Haskell didn't have a facility for expressing typeclasses in terms of higher-kinded types at all. This was developed by Mark P. Jones[2] while he was working on an implementation of Haskell called *Gofer*. This work generalized typeclasses 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 typeclasses, 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 typeclass should abide by two basic laws. Undertanding these laws is critical for understanding Functor and writing typeclass instances that are composable and easy to reason about.

Identity

The first law is the law of identity:

```
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 didn't return a new value in the `a -> b` function mapped over the structure, then nothing should've changed:

```
Prelude> fmap id "Hi Julie"
"Hi Julie"
Prelude> id "Hi Julie"
```

---

[2]A system of constructor classes: overloading and implicit higher-order polymorphism
`http://www.cs.tufts.edu/~nr/cs257/archive/mark-jones/fpca93.pdf`

```
"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 fmapped them 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 typeclass Functor, but that is all we know about its type from this definition. As we've seen with typeclass-constrained

polymorphism, this still allows it to be any type that has an instance of
Functor. The core operation that this typeclass 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
definining 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 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 dosey-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> fmap id WhatThisIsCalled == id WhatThisIsCalled
False
```

This certainly does not abide by the identity law. It is 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 just 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 just 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 nice special name too, but there isn't a widely agreed upon typeclass.

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 composed the functions ahead of time and then lifted the composed function all together. Maintaining this property is about preserving 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 okay
instance Functor CountingBad where
  fmap f     (Heisenberg n a) = Heisenberg (n+1) (f a)
-- (a -> b)    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 `fmap`s here does not produce the same results, so the composition law doesn't hold:

```
Prelude> let oneWhoKnocks = Heisenberg 0 "Uncle"
Prelude> fmap (++" Jesse") oneWhoKnocks
Heisenberg 1 "Uncle Jesse"
Prelude> fmap ((++" Jesse") . (++" lol")) oneWhoKnocks
Heisenberg 1 "Uncle lol Jesse"
```

So far it seems OK, but what if we compose the two concatenation functions separately?

```
Prelude> fmap (++ " Jesse") . fmap (++ " lol") $ oneWhoKnocks
Heisenberg 2 "Uncle lol Jesse"
```

Or to make it look more like the law:

```
Prelude> let f = (++" Jesse")
Prelude> let 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

```
fmap (f . g)  ==  fmap f . fmap g
```

does not hold. So how do we fix it?

```haskell
data CountingGood a =
  Heisenberg Int a
  deriving (Eq, Show)

-- Totes cool.
instance Functor CountingGood where
  fmap f (Heisenberg n a) = Heisenberg (n) (f a)
```

Just 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 all is going on with a minimum of fuss.

We begin with a simple helper function:

```
Prelude> :t const
const :: a -> b -> a
Prelude> let 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> let 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> let 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> let lms = [Just "Ave", Nothing, Just "woohoo"]

Prelude> let 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 List (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> (fmap . fmap . fmap) replaceWithP lms
[Just "ppp",Nothing,Just "pppppp"]
```

Let's review what we just saw, but with an X-ray turned on:

```
-- lms ~ List (Maybe String)

Prelude> let lms = [Just "Ave", Nothing, Just "woohoo"]

Prelude> replaceWithP lms
'p'

Prelude> :t replaceWithP lms
replaceWithP lms :: Char


-- In:
replaceWithP lms

-- replaceWithP's input type is:
List (Maybe String)

-- The output type is Char

-- So applying
replaceWithP

-- to
lms

-- accomplishes
List (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:
List (Maybe String) -> List Char

-- List 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 List Char or [Char]

-- The output type is Char

-- So applying
(fmap . fmap) replaceWithP

-- to
lms

-- accomplishes
List (Maybe String) -> List (Maybe Char)
```

Wait, how does that even typecheck? It may not seem obvious at first how (`fmap . fmap`) could typecheck. 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> (fmap . fmap . fmap) replaceWithP lms
[Just "ppp",Nothing,Just "pppppp"]

Prelude> :t (fmap . fmap . fmap) 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're going to summarize here before we move on:

```
Prelude> fmap replaceWithP lms
"ppp"

Prelude> (fmap . fmap) replaceWithP lms
[Just 'p',Nothing,Just 'p']

Prelude> (fmap . fmap . fmap) 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're understanding everything we've done so far. If not, play with it until it starts to feel 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> let ha = Just ["Ha", "Ha"]
Prelude> let lmls = [ha, Nothing, Just []]

Prelude> (fmap . fmap) replaceWithP lmls
[Just 'p',Nothing,Just 'p']

Prelude> (fmap . fmap . fmap) replaceWithP lmls
[Just "pp",Nothing,Just ""]

Prelude> (fmap . fmap . fmap . 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"]

-- Just making 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
```

```haskell
liftedReplace :: Functor f => f a -> f Char
liftedReplace = fmap replaceWithP
```

But we can assert a more specific type for `liftedReplace`!

```haskell
liftedReplace' :: [Maybe [Char]] -> [Char]
liftedReplace' = liftedReplace
```

The `[]` around Char is the *f* of `f Char`, or the structure we lifted over. The *f* of `f a` is the outermost `[]` in [Maybe [Char]]. So, `f ~ []` 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)
```

```haskell
twiceLifted :: (Functor f1, Functor f) =>
               f (f1 a) -> f (f1 Char)
twiceLifted = (fmap . fmap) replaceWithP
```

```
-- Making it more specific
```

```haskell
twiceLifted' :: [Maybe [Char]] -> [Maybe Char]
twiceLifted' = twiceLifted
-- f  ~ []
-- f1 ~ Maybe
```

Thrice?

```haskell
-- Prelude> :t (fmap . fmap . fmap) replaceWithP
-- (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:

```haskell
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, run **main** to see what output results. Then, modify the types and 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!

Intermission: Lifting Exercises

Add **fmap**, parentheses, and function composition to the expression as needed for the expression to typecheck and produce the expected result. It may not always need to go in the same place, so don't get complacent.

1. ```haskell
   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   Mapping over the structure to transform 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 fmapping.

We know that the goal of fmapping 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 fmapped 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 tranform 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've demonstrated this by applying the type constructor to concrete types; however, you can just as well 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 was 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 typechecker 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 either, 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.

```haskell
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

-- remember, names don't mean anything beyond
-- their relationships to each other.
:: (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:

```haskell
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:

```haskell
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:

```haskell
fmap id      = id
fmap (p . q) = (fmap p) . (fmap q)
```

We can turn those into the following QuickCheck properties:

```haskell
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 provided concrete instances, we can now run these to test them.

```
Prelude> quickCheck $ \x -> functorIdentity (x :: [Int])
+++ OK, passed 100 tests.

Prelude> let li x = functorCompose (+1) (*2) (x :: [Int])

Prelude> quickCheck li
+++ OK, passed 100 tests.
```

Groovy.

Making QuickCheck generate functions too

QuickCheck happens to offer the ability to generate functions. There's a different but related typeclass called CoArbitrary, this covers the function

argument type where Arbitrary is used for the function result type. If you're curious about this, take a look at the Function module in the QuickCheck library to see how functions are generated from a datatype that represents patterns in function construction.

```
{-# LANGUAGE ViewPatterns #-}

import Test.QuickCheck
import Test.QuickCheck.Function

functorCompose' :: (Eq (f c), Functor f) =>
                        f a
                -> Fun a b
                -> Fun b c
                -> Bool
functorCompose' x (Fun _ f) (Fun _ g) =
  (fmap (g . f) x) == (fmap g . fmap f $ x)
```

There are a couple things going on here. One is that we needed to import a new module from QuickCheck. Another is that we're pattern matching on the `Fun` value that we're asking QuickCheck to generate. The underlying `Fun` type is essentially a product of the weird function type and an ordinary Haskell function generated from the weirdo. The weirdo QuickCheck-specific concrete function is a function represented by a datatype which can be inspected and recursed. We only want the second part, the ordinary Haskell function, so we're pattern-matching that one out.

```
Prelude> type IntToInt = Fun Int Int
Prelude> type IntFC = [Int] -> IntToInt -> IntToInt -> Bool
Prelude> quickCheck (functorCompose' :: IntFC)
+++ OK, passed 100 tests.
```

Note of warning, you can't print those Fun values, so **verboseCheck** will curse Socrates and spin in a circle if you try it.

16.10 Intermission: Exercises

Implement Functor instances for the following datatypes. Use the QuickCheck properties we just 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 Functors work, 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:

```haskell
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:

```haskell
someFunc Nothing = Nothing
```

Then they're just applying some function to the value if it's a Just:

```haskell
someFunc (Just x) = Just $ someOtherFunc x
```

What happens if we use `fmap`?

```haskell
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?

```haskell
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 contract 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! In fact, we can query the type of the expressions in GHCi and see for ourselves the more generic type:

```
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:

```haskell
-- ``lifted'' because they've been 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"]
```

Short Exercise

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.

```haskell
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:

```haskell
applyIfJust :: (a -> b) -> Maybe a -> Maybe b
```

Either

The Maybe type solves a lot of problems for Haskellers. But it doesn't solve all of them. As we saw in a previous chapter, sometimes we want to preserve the reason *why* a computation failed rather than only the information that it failed. And for that, we use Either.

By this point, you know that Either has a Functor instance ready-made for use by grateful programmers. So let's put it to use. We'll stick to the same pattern we used for demonstrating Maybe, for the sake of clarity:

```haskell
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 obvious "leave the error value alone" case:

```haskell
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:

```haskell
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:

```haskell
-- 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 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 also already has a Functor instance.

```haskell
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 writing the following function.

```haskell
applyIfSecond :: (a -> b) -> (Sum e) a -> (Sum e) b
```

2. Why is a Functor instance that applies the 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> let a = const 1
Prelude> a 1
1
Prelude> a 2
1
Prelude> a 3
1
Prelude> a "blah"
1
Prelude> a id
1
```

With a similar concept in mind, there is the Constant datatype. We'll write a newtype called `Constant` so as not to conflict with Prelude:

```
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, though, 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`, just 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> getConstant $ fmap (const 2) (Constant 3)
3
Prelude> getConstant $ fmap (const "blah") (Constant 3)
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 `fmap`'d 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> let separate = fmap (const 3) . fmap (const 5)
Prelude> let fused = fmap ((const 3) . (const 5))
Prelude> getConstant $ separate $ (Constant "WOOHOO")
"WOOHOO"
Prelude> getConstant $ fused $ (Constant "Dogs rule")
"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, Const is a lawful Functor.

16.13   More structure, more functors

```haskell
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?

```haskell
instance Functor (Wrap f) where
  fmap f (Wrap fa) = Wrap (f fa)
```

```haskell
instance Functor (Wrap f) where
  fmap f (Wrap fa) = Wrap (fmap f fa)
```

```haskell
instance Functor f => Functor (Wrap f) where
  fmap f (Wrap fa) = Wrap (fmap f fa)
```

And if we load up the final instance which should work, we can use this wrapper type:

```haskell
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> let n = 1 :: Integer
Prelude> fmap (+1) (Wrap n)



Couldn't match expected type 'f b' with actual type 'Integer'
Relevant bindings include
     it :: Wrap f b (bound at <interactive>:8:1)
In the first argument of 'Wrap', namely 'n'
In the second argument of 'fmap', namely '(Wrap n)'
```

The number by itself doesn't offer the additional structure needs for Wrap to work as a Functor. It's expecting to be able to fmap over some f independent of an a and this just isn't the case with any type constant such as Integer.

16.14   IO Functor

We've seen the `IO` type in the modules and testing chapters already, but we weren't doing 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 typeclasses `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 enter 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 my entering 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 applied `const ()` to the contents of the `IO Int`. If we
ignore the presence of IO, it's as if we did this:

```
Prelude> let 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 evaluated was evaluated
for effects and because the unit value cannot communicate anything. We
can use the return function (seen earlier, explained later) to "put" a unit
value in IO and reproduce this behavior of GHCi's.

```
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 what 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 ver-
bose 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 talked about Functor as a means of lifting functions over structure so
that we may transform only the contents, leaving the structure alone. What
if we wanted to transform only the *structure* 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
```

So in a sense, we're doing the opposite of what a Functor does. We're transforming the structure, preserving the values as they were. 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 was given.

Syntactically, it lets us avoid talking about a in the type of Nat — which is what we want, we shouldn't *have* any specific information about the contents of f and g because we're supposed to be only 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 Rank2Types or RankNTypes, it won't work:

```
Prelude> type Nat f g = forall a . f a -> g a

Illegal symbol '.' in type
Perhaps you intended to use RankNTypes or a
similar language extension to enable
explicit-forall syntax: forall <tvs>. <type>
```

If turn on Rank2Types or RankNTypes, it works fine:

```
Prelude> :set -XRank2Types
Prelude> type Nat f g = forall a . f a -> g a
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?

```haskell
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]

-- But this will too if we tell it
-- '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 this isn't a good way to think about natural transformation. Part of software 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 something clever, just 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 in Haskell are unique for a given datatype

In Haskell, Functor instances will be unique for a given datatype. We saw that this isn't true for Monoid; however, we use newtypes to avoid confusing different Monoid instances for a given 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:

```haskell
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:

```haskell
{-# 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 actually 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 if a valid Functor can be written for the datatype provided.

1. ```haskell
   data Bool =
       False | True
   ```

2. ```haskell
   data BoolAndSomethingElse a =
       False' a | True' a
   ```

3. ```haskell
   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.

   ```haskell
   newtype Mu f = InF { outF :: f (Mu f) }
   ```

5. Again, just follow the kinds and ignore the unfamiliar parts

   ```haskell
   import GHC.Arr
   ```

   ```haskell
   data D =
     D (Array Word Word) Int Int
   ```

Rearrange the arguments to the type constructor of the datatype so the Functor instance works.

1. ```haskell
   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.
```haskell
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.
```haskell
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:

```haskell
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.
```haskell
data Quant a b =
    Finance
  | Desk a
  | Bloor b
```

2. No, it's not interesting by itself.

```haskell
data K a b =
  K a
```

3. *{-# LANGUAGE FlexibleInstances #-}*

```haskell
newtype Flip f a b =
  Flip (f b a)
  deriving (Eq, Show)

newtype K a b =
  K a

-- should remind you of an
-- instance you've written before
instance Functor (Flip K a) where
  fmap = undefined
```

4.
```haskell
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?

```haskell
data LiftItOut f a =
  LiftItOut (f a)
```

6.
```haskell
data Parappa f g a =
  DaWrappa (f a) (g a)
```

7. Don't ask for more typeclass instances than you need. You can let GHC tell you what to do.

```haskell
data IgnoreOne f g a b =
  IgnoringSomething (f a) (g b)
```

8.
```haskell
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 a Goat-Lord, 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.[3]

```
data TalkToMe a =
    Halt
  | Print  String a
  | Read  (String -> a)
```

---

[3]Thanks to Andraz Bajt for inspiring this exercise.

16.18 Definitions

1. A *higher-kinded type* is a type which itself takes types as arguments and potentially mentions them in the definition of the datatype.

2. *Functor* is a mapping between categories. In Haskell, this manifests as a typeclass which lifts a function between two types over two new types. This conventionally implies some notion of a function which 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 typeclass.

```haskell
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.

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? In this book, we most commonly use the phrase *lift over* (later, in Monad, *bind over* for reasons that will hopefully become clear in time), but it's important to understand that this is a metaphor, and spatial metaphors can, at times, obscure as much as they clarify. When 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 to 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 both cases, the function we're lifting is the same. In the first case, we lift that function into a Maybe context in order to apply it; in the second case, into a list context. It can be helpful to think of it in terms of lifting the function into the context, because it's the context we've lifted the function into that determines how the function will get applied (to just 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.

You might find one or the other way of talking about lifting more or less helpful, depending on the context. Try to keep in mind that it's a metaphor, not a precise definition, and *follow the types* rather than getting too caught up in the 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 have 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.
   `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.
   `haskellforall.com`

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've seen two common algebras that are used as typeclasses in Haskell. 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 that structure unaltered.

We come now to Applicative. Applicative is a monoidal functor. No, no, stay with us. The `Applicative` typeclass 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 typeclass 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 typeclass declaration is that the f that represents the structure, similar to Functor, is itself constrained by the Functor typeclass:

```
class Funstor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

```
-- Could call <*> tie-fighter or "ap" (short for apply)
```

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 embeds 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 core operation of this typeclass 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 just 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 just `fmap`, you are correct. It is basically the same as `fmap` only with an Applicative typeclass constraint instead of a Functor one. Since all Applicative instances are also functors, though, this is a distinction without much significance.

Similarly you can see that `liftA2` and `liftA3` are just `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 simple 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 appears to be quite small and innocuous. We now have an *f* in front of our function `(a -> b)`. But 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 it 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 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 which exists merely 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 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 just 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. Now,
they will be definitely the same type, because the *f* must be the same type
throughout. In fact, if we just 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 merely mapping over with Functor. Let's consider a few familiar examples to examine what this means:

```
-- List
[(*2), (*3)] <*> [4, 5]

=

[2 * 4, 2 * 5, 3 * 4, 3 * 5]

-- reduced

[8,10,12,15]
```

So what was `(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 was just the List Functor.

But "list-ness" isn't the only structure we can enrich our functions with — not even close! Thinking of Functor and Applicative as things you mostly do with lists is an easy mistake to make as you're learning. But 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 just 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 two-tuple ignores the first value
inside the tuple:

```
Prelude> fmap (+1) ("blah", 0)
("blah",1)
```

But the Applicative for the two-tuple demonstrates the monoid in Applica-
tive 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 two-tuple, we don't need a Monoid for the *b* because we're using function application to produce the *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 didn't apply any function, but they have combined 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 has been 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, we just need some way of combining or choosing our values.

Tuple Monoid and Applicative side by side

Squint if you can't see it.

```haskell
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 really monoidal functors, be careful about taking it too literally. For one thing, Monoid and Applicative instances aren't required or guaranteed to have the same monoid of structure, and the functorial part may actually change things. 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:

```haskell
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)
```

In the next chapter, we're going to 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
```

What's the List applicative do?

Previously with List functor, we were mapping 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 are mapping 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 happened with that expression we just tested? 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 merely 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]

-- first we 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 become 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> lookup 3 [(3, "hello")]
Just "hello"
Prelude> fmap length $ lookup 3 [(3, "hello")]
Just 5
Prelude> let c (x:xs) = toUpper x:xs
Prelude> fmap c $ lookup 3 [(3, "hello")]
Just "Hello"
```

---

[1]`en.wikipedia.org/wiki/Cartesian_product`

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 a Map version of **lookup** along with **fromList** to accomplish the same thing with that data structure:

```
Prelude> fmap c $ Data.Map.lookup 3 (fromList [(3, "hello")])
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 file, 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 simple 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 resulting string of the first example.

Short Exercises

In the following exercises you will need to use the following terms to make the expressions type-check:

    1. pure

2. `(<$>)`
   ```
   -- or fmap
   ```

3. `(<*>)`

Make the following expressions type-check.

1. ```haskell
   added :: Maybe Integer
   added = (+3) (lookup 3 $ zip [1, 2, 3] [4, 5, 6])
   ```

2. ```haskell
   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. ```haskell
   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 these typeclasses 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 =>
(<*>) ::          f (a -> b) ->          f a ->          f b
(<*>) :: Identity (a -> b) -> Identity a -> Identity b

pure :: a ->          f a
pure :: a -> Identity a
```

Why would we use Identity just to introduce some structure? What is the meaning of all this?

```
Prelude> const <$> [1, 2, 3] <*> [9, 9, 9]
```

```
[1,1,1,2,2,2,3,3,3]
Prelude> const <$> Identity [1, 2, 3] <*> Identity [9, 9, 9]
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

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 this 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-life 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 just keep trying.

This datatype is like the **const** function in that it takes two arguments but one of them just 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 just doesn't happen.

Specializing the types

All right, so here's what the types will look like:

```
-- f ~ Constant e

(<*>) ::            f (a -> b) ->            f a ->           f b
(<*>) :: Constant e (a -> b) -> Constant e a -> Constant e b

pure :: a ->            f a
pure :: a -> Constant 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 actually going on:

```
*Main> Constant (Sum 1) <*> Constant (Sum 2)
Constant {getConstant = Sum {getSum = 3}
Prelude> Constant undefined <*> Constant (Sum 2)
Constant (Sum {getSum = *** Exception: Prelude.undefined
*Main> pure 1
1
*Main> 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 just throwing away a function application. We
know it seems somewhat crazy, but we promise there are really times real
coders do this in real code. Pinky swear.

Exercise

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 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 of the function to be applied being a Nothing case.

Specializing the types

Here's what the type looks like when we're using Maybe as our *f* structure:

```
  -- f ~ Maybe
(<*>) ::       f (a -> b) ->      f a ->      f b
(<*>) :: Maybe (a -> b) -> Maybe a -> Maybe b

pure :: a ->      f a
pure :: a -> Maybe 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`, where the Maybe is because our inputs might
be invalid:

```haskell
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`:

```haskell
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 here 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 sort of a roadblock. Can you see the problem?

```
Prelude> let maybeAddy = (mkAddress "old macdonald's")
Prelude> :t fmap (fmap Person (mkName "Babe")) maybeAddy

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")) maybeAddy
```

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> (fmap Person (mkName "Babe")) <*> maybeAddy
Just (Person (Name "Babe") (Address "old macdonald's"))
```

Nice, right? A little ugly though. As it happens, Control.Applicative gives us an infix alias for `fmap` called `<$>`. You should probably just call it "fmap", but the authors won't tell anybody if you're saying "cash-money" in your head.

```
Prelude> Person <$> mkName "Babe" <*> maybeAddy
Just (Person (Name "Babe") (Address "old macdonald's"))
```

Much better! 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 that.

We can now use a much shorter and nicer 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 added new fields as well.

Breaking down the example we just saw

We're going to give the Functor and Applicative instances for Maybe the same treatment we gave fold. This will be a bit long. It is possible that some of this will seem like too much detail; read it to whatever depth you feel you need to. It will sit here, patiently waiting to see if you ever need to come back and read it more closely.

Maybe Functor and the Name constructor

```haskell
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:

```haskell
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 the chapter on folds.

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 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 doesn't > 25, so:
validateLength 25 "babe" = Just "babe"
```

Now we're going to replace `validateLength` applied to 25 and "babe" with what it evaluated 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 *fa* is String:

```
(a -> b) -> f a -> f b
```

```
:t Name        :: (String -> Name)
:t Just "babe" :: Maybe String
```

```
    (a -> b)     -> f    a      -> f    b
(String -> Name) -> Maybe String -> Maybe 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 `Person (Name "babe")` is awaiting another argument, the
address, so it's a partially applied function. That's our `(a -> b)` in the
type of Applicative's `(<*>)`. The *f* wrapping our `(a -> b)` is the Maybe
which 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

Maybe (Address -> Person) -> Maybe Address -> Maybe 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 of the Maybe Applicative. 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.

```haskell
-- 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

```haskell
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
```

```haskell
-- Validating to get rid of empty strings, 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!

```haskell
-- you'll need to import this if you have GHC <7.10

import Control.Applicative

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:

```haskell
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> let cow1 = Cow <$> noEmpty "Bess"

Prelude> :t cow1
cow1 :: Maybe (Int -> Int -> Cow)

Prelude> let cow2 = cow1 <*> noNegative 1

Prelude> :t cow2
cow2 :: Maybe (Int -> Cow)

Prelude> let cow3 = cow2 <*> noNegative 2

Prelude> :t cow3
cow3 :: Maybe Cow
```

Then with liftA3:

```
Prelude> let cow1 = liftA3 Cow

Prelude> :t cow1
cow1 :: Applicative f => f String -> f Int -> f Int -> f Cow

Prelude> let cow2 = cow1 (noEmpty "blah")

Prelude> :t cow2
cow2 :: Maybe Int -> Maybe Int -> Maybe Cow

Prelude> let cow3 = cow2 (noNegative 1)

Prelude> :t cow3
cow3 :: Maybe Int -> Maybe Cow

Prelude> let 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 just 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 how this in the following specialization of the apply function (<*>):

```
(<*>) :: Applicative f => f (a -> b) -> f a -> f b

f ~ []

maybeApply :: Maybe (a -> b) -> Maybe a -> Maybe b
maybeFmap  ::       (a -> b) -> Maybe a -> Maybe b

-- maybeFmap is just fmap's type specialized to Maybe
```

You can test these specializations (more concrete versions) of the types in the REPL:

```
Prelude> :t (<*>)
(<*>) :: Applicative f => f (a -> b) -> f a -> f b

Prelude> :t (<*>) :: Maybe (a -> b) -> Maybe a -> Maybe b
(<*>) :: Maybe (a -> b) -> Maybe a -> Maybe b
  :: Maybe (a -> b) -> Maybe a -> Maybe b

Prelude> :t fmap
fmap :: Functor f => (a -> b) -> f a -> f b

Prelude> :t fmap :: (a -> b) -> Maybe a -> Maybe b
fmap :: (a -> b) -> Maybe a -> Maybe b
  :: (a -> b) -> Maybe a -> Maybe b
```

If you make any mistakes, the REPL will let you know:

```
Prelude> :t fmap :: (a -> b) -> Maybe a -> f b

Couldn't match type 'f1' with 'Maybe'
'f1' is a rigid type variable bound by
  an expression type signature:
    (a1 -> b1) -> Maybe a1 -> f1 b1

Expected type: (a1 -> b1) -> Maybe a1 -> f1 b1
```

```
   Actual type: (a1 -> b1) -> f1 a1 -> f1 b1
In the expression: fmap :: (a -> b) -> Maybe a -> f b

Prelude> :t (<*>) :: Maybe (a -> b) -> Maybe a -> f b

Couldn't match type 'f1' with 'Maybe'
'f1' is a rigid type variable bound by
  an expression type signature:
    Maybe (a1 -> b1) -> Maybe a1 -> f1 b1

Expected type: Maybe (a1 -> b1) -> Maybe a1 -> f1 b1
  Actual type: f1 (a1 -> b1) -> f1 a1 -> f1 b1

In the expression:
  (<*>) :: Maybe (a -> b) -> Maybe a -> f b
```

Exercise

Given the function and values provided, use **(<$>)** from Functor, **(<*>)** and **pure** from the Applicative typeclass to fill in missing bits of the broken code 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 then composing

them should be the same. We're using the composition operator as a prefix instead of the more usual infix, and 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 categories. 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
```

Because the "structure" that **pure** is providing there isn't really 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.

```haskell
pure (+1) <*> pure 1 :: Maybe Int
```

```haskell
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:

```haskell
(+1) 1
```

be equal this time around?

Here are a couple more examples to try out:

```haskell
pure (+1) <*> pure 1 :: [Int]
```

```haskell
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:

   ```haskell
   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 ($    y)  <*>   u
  f   (a -> b)       f a

pure  ($    2) <*> Just (+ 2)
-- f  (a -> b)  ->   f    a   ->

-- the y is the b of the
-- f (a -> b)
-- so it goes something like this:

Just ((+ 2) $ 2)
-- f    a  -> b  ->

-- equals
Just  4
-- f  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 those four laws. This keeps your code composable and free of unpleasant surprises.

17.7   You knew this was coming

QuickChecking the Applicative laws! You should have got the gist of how to write properties based on laws, so we're going to use a pre-existing library this time. Conal Elliott has a nice library called *checkers* on Hackage and Github which provides some nice pre-existing properties and utilities for QuickCheck.

After installing *checkers*, we can reuse the existing properties for validating Monoids and Functors to revisit what we did previously.

```haskell
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

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` which 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 it doesn't actually *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 was able to falsify left and right identity for `Bull`. Now lets test a pre-existing Applicative instance, such as list and maybe. The type for the `TestBatch` which 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*, *b* to *c*) embedded in that structure, and that it wants EqProp instances. That's all well and good, but we can ignore that.

```
--  :: (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
```

We just care about `m (a, b, c) -> TestBatch`. We could pass an actual value giving us our Applicative structure and three values which could be of different type, 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> quickBatch $ applicative [("b", "w", 1 :: Int)]

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 to not have an ambiguous type. We would've had to specify that outside of GHCi. In the following example we'll use a bottom to fire the typeclass dispatch:

```
Prelude> let trigger = undefined :: [(String, String, Int)]
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

No instance for (Num a0) arising from a use of 'it'
The type variable 'a0' is ambiguous
Note: there are several potential instances:
   ... some blather that mentions Num ...

Prelude> 1 <> (2 :: Sum Integer)
Sum {getSum = 3}
```

Prelude doesn't provide this Monoid for us, so we must define it ourselves.

```haskell
module Apl1 where

import Control.Applicative
import Data.Monoid
import Test.QuickCheck
import Test.QuickCheck.Checkers
import Test.QuickCheck.Classes

-- unfortunate orphan instances. 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> quickBatch $ monoid (ZipList [1 :: Sum Int])

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 the List Applicative. 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> let functions = Cons (+1) (Cons (*2) Nil)
Prelude> let values = Cons 1 (Cons 2 Nil)
Prelude> functions <*> values
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 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> flatMap (\x -> [x, 9]) [1, 2, 3]
[1,9,2,9,3,9]
```

Applicative instances, unlike Functors, are not guaranteed to have a unique implementation for a given datatype.

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.

```haskell
data List a =
    Nil
  | Cons a (List a)
  deriving (Eq, Show)

take' :: Int -> List a -> List a
take' = undefined

instance Functor List where
  fmap = undefined

instance Applicative List where
  pure = undefined
  (<*>) = undefined

newtype ZipList' a =
  ZipList' (List 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
```

A couple hints: think infinitely. 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 looking for inspiration from these functions, not to be able to directly
reuse them as you're using a custom List type and 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 in-
stead 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

(<*>) ::           f (a -> b) ->           f a ->          f b
(<*>) :: Either e (a -> b) -> Either e a -> Either e b

pure :: a ->          f a
pure :: a -> Either e a
```

Either versus Validation

Often the interesting part of an Applicative is wherever the "monoidal" in
"monoidal functor" is coming from. One byproduct of this is that just as you
can have more than one valid Monoid for a given datatype, *unlike 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 that differs only in the Applicative instance called
Validation:

```
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 which can go between Validation and Either
values interchangeably:

```haskell
-- Remember when we mentioned natural transformations?
-- Both of these functions are natural transformations.

validToEither :: Validation e a -> Either e a
validToEither (Failure err) = Left err
validToEither (Success a) = Right a

eitherToValid :: Either e a -> Validation e a
eitherToValid (Left err) = Failure err
eitherToValid (Right a) = Success a

eitherToValid . validToEither == id
validToEither . eitherToValid == 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 typeclass to combine them. Often this'll just be a list or set of errors but you can do whatever you want.

```haskell
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 occurred, not just the first one.

Exercise

Write the Either Applicative that short-circuits on any error values. Sum
and Validation are both just alternative names for Either, but you'll be
giving them different Applicative instances. See above for an idea of how
Validation should behave. Use the *checkers* library.

```haskell
data Sum a b =
    First a
  | Second b
  deriving (Eq, Show)

data Validation e a =
    Error e
  | Success a
  deriving (Eq, Show)

instance Functor (Sum a) where
  fmap = undefined

instance Applicative (Sum a) where
  pure = undefined
  (<*>) = undefined

-- same as Sum/Either
instance Functor (Validation e) where
  fmap = undefined

-- This is different
instance Monoid e =>
         Applicative (Validation e) where
  pure  = undefined
  (<*>) = undefined
```

Your hint for this one is that you're writing the following functions:

```haskell
applyIfBothSecond :: (Sum e) (a -> b)
                  -> (Sum e) a
                  -> (Sum e) b

applyMappendError :: Monoid e =>
                     (Validation e) (a -> b)
                  -> (Validation e) a
                  -> (Validation e) b
```

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.

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 applicative instances for the following datatypes. Confused? Write out what the type should be. Use the *checkers* library to validate the instances.

1. **newtype Identity** a = **Identity** a **deriving Show**

2. **data Pair** a = **Pair** a a **deriving Show**

3. This should look familiar.

   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

Combinations

Remember the vowels and stops exercise in folds? Reimplement the combos
function using **liftA3** from Control.Applicative.

```haskell
import Control.Applicative (liftA3)

stops, vowels :: String
stops = "pbtdkg"
vowels = "aeiou"

combos :: [a] -> [b] -> [c] -> [(a, b, c)]
combos = undefined
```

17.10   Definitions

1. Applicative is a typeclass in Haskell which you could think of as the Jonathan Taylor Thomas to Functor and Monad's Taran Noah Smith and Zachery Ty Bryan. Applicative can be thought of characterizing monoidal functors in Haskell. For a Haskeller's purposes, it's a way to functorially apply a function which is embedded in 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; Nick Partridge; Validation library
   http://hackage.haskell.org/package/validation

2. Conor McBride; Ross Paterson; Applicative Programming with Effects
   http://staff.city.ac.uk/~ross/papers/Applicative.html

3. Jeremy Gibbons; Bruno C. d. S. Oliveira; Essence of the Iterator Pattern

4. Ross Paterson; Constructing Applicative Functors
   http://staff.city.ac.uk/~ross/papers/Constructors.html

5. Sam Lindley; Philip Wadler; Jeremy Yallop; Idioms are oblivious, arrows are meticulous, monads are promiscuous.

   Note: Idiom means applicative functor and is a useful search term for published work on applicative functors.

17.12 Answers

Short Exercise

1. `(+3) <$> (lookup 3 $ zip [1, 2, 3] [4, 5, 6])`
   `Just 9`

2. `(,) <$> (lookup 3 $ zip [1, 2, 3] [4, 5, 6])`
   `    <*> (lookup 2 $ zip [1, 2, 3] [4, 5, 6])`
   `Just (6,5)`

Chapter 18

Monad

There is nothing so practical as
a good theory

_____

Phil Wadler, quoting 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 standard for Haskell does use monad 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

- define `Monad`, its operations and laws;

- look at several examples of monads in practice;

- write the `Monad` instances for various types;

- address some misinformation about monads.

18.2   We're very sorry, but 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 over some structure over some structure and then mappends the two bits of structure. So you can think of monads as just another way of applying functions over structure, with a couple of additional features. We'll

---

[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`

get to those features in a moment. For now, let's check out the typeclass 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:

```
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 typeclass 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 just fine:

```
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 just a convenience. `Functor`, `Applicative`, and `Monad` instances for a given type should have the same core behavior.

We'll explore the relationship between these classes more completely in just a bit, but as part of understanding the typeclass 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` typeclass 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`, pretty easily: 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 the Modules chapter, 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. Basically 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, contains — the things that are special about `Monad`.

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 just another name for `pure` from `Applicative`.

We also noted (and will see more clearly soon) that it also isn't `>>` which 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 by fmapping a function of type `(a -> m b)` to make it more like `>>=`, and it will work just fine. 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> let 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 knew from our types that we'd end up with an `f (f b)` — that is, an extra layer of structure, and now we have a result of nested lists. What if we just wanted `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:

```haskell
import Control.Monad (join)

join :: Monad m => m (m a) -> m a

-- compare

concat ::          [[a]] -> [a]
```

It's also somewhat novel that we can inject more structure via our function application, where applicatives and fmaps 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 truly makes `Monad` special. And it's by putting that `join` function together with the mapping function that we get `bind`, also known as `>>=`.

So how do we get bind?

The answer is the exercise   Write `bind` in terms of `fmap` and `join`.

Fear is the mind-killer, friend. You can do it.

```haskell
bind :: Monad m => (a -> m b) -> m a -> m b
bind = undefined
```

What Monad is not

Since `Monad` is somewhat abstract and can be quite 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 can do, and to a lesser extent, a wrong notion of what `IO` is all about.

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 typeclass describes a specific relationship between elements in a domain and defines some operations over them. When we refer to something as "a monad," we're using that the same way we talk about "a monoid," or "a functor." None of those are values.

4. About strictness. The monadic operations of `bind` and `return` are nonstrict. 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` typeclass 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 really 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 really 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. So, you may still see them sometimes. We'll take a short tour of them, 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, that is just `fmap` with a different typeclass 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 out:

```
Prelude> liftA2 (,) (Just 3) (Just 5)
Just (3,5)
```

```
Prelude> liftM2 (,) (Just 3) (Just 5)
Just (3,5)
```

You may remember way back in Lists, 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 hap-
pens:

```
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 the Modules chapter. 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 performing `IO` actions. This section is going to talk about why **do** is just 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> putStrLn "Hello, " >> putStrLn "World!"
Hello,
World!
```

```
Prelude> putStrLn "Hello, " *> putStrLn "World!"
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:

```haskell
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:

```haskell
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 just use `>>=` which passes it directly.

When fmap alone isn't enough

Note that if you try to `fmap putStrLn` 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 gave it didn't print, although it seems like it should have due to the `putStrLn` being mapped over the `getLine`. We evaluated the `IO` action that requests input, but not the one that prints it. So, what happened?

Well, let's start with the types. The type of what you just tried 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 didn't work. First, `getLine` performs `IO` to get a `String`:

```
getLine :: IO String
```

And `putStrLn` takes a `String` argument, performs `IO`, 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 `IO` that `getLine` performs. Perhaps this looks familiar from our demonstration of what happens when you use `fmap` to map a function that of `(a -> m b)` instead of just `(a -> b)` — that is what's happening here. So 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 ())
```

Okay...so, which `IO` is which, and why does it ask for input but not print what we typed 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 typed in.

2. This inner `IO` structure represents the effects that would be performed *if* `putStrLn` was 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 really), consider the following:

```
Prelude> let printOne = putStrLn "1"
Prelude> let printTwo = putStrLn "2"
Prelude> let 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 just `fmap 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 ()
```

MONAD: IT'S GOT WHAT CODERS CRAVE.

What `join` did 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 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 just 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 are 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:

```haskell
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:

```haskell
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.")
```

Here is the same example as above, only this time with parens:

```haskell
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 typeclasses 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

-- or more syntactically common
(>>=) ::             [a] -> (a -> [b])   -> [b]

-- same 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:

```haskell
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` 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:

```haskell
Prelude> twiceWhenEven [1..3]
[1,4,4,9]
```

Now try this:

```haskell
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). Was the result what you expected? 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

Now we come to a more exciting demonstration of what we can do with our newfound power.

Specializing the types

Tis the season for examining the types:

```
-- 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 intuitions above for `do` syntax and the list monad; here we'll be quite explicit about what's happening, and by the time we get to the `Either` demonstration below, it should be clear. Let's get started:

```haskell
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, 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 `Monad`, then we'll see why we can't do this with `Applicative`:

```
-- Do syntax isn't just for IO.

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!

```
-- Stack up dem nested lambdas.

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
  return (zed a b c)
```

You can rewrite it using Applicative. On the other hand, if you have something like this:

```
doSomething = do
  a <- f
  b <- g
  c <- h
  zed a b c
```

You're going to need Monad because `zed` is producing more monadic structure, and you'll need `join` to crunch that 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 :: Maybe Integer
f = Just 1

g :: Maybe String
g = Just "1"

h :: Maybe Integer
h = Just 10191
```

```haskell
zed :: a -> b -> c -> (a, b, c)
zed = (,,)

doSomething = do
  a <- f
  b <- g
  c <- h
  return (zed a b c)


zed' :: Monad m => a -> b -> c -> m (a, b, c)
zed' a b c = return (a, b, c)

doSomething' = do
  a <- f
  b <- g
  c <- h
  zed' a b c
```

The long and short of it:

1. With the `Maybe Applicative`, each `Maybe` computation fails or succeeds independently of each other. You're just 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 in the guts of this code and how binding over `Maybe` values works.

For once, this example instance is what's actually in GHC's base library at time of writing:

```haskell
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?

```haskell
-- Proceeding outermost to innermost.

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 produced the value `Just "Bess"`; however, `nammy` will be just the
string and not also the `Maybe` structure because `>>=` passes a to the function
it binds over the monadic value, not ma. 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 the **bind** function 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 just be thankful that cow was full of `Maybe` values and not tripe. Moving along, we're going to demonstrate use of the `Either Monad`, step back a bit, and let your intuitions and what you learned about `Maybe` guide you through.

Specializing the types

As always, we present the types:

```haskell
-- 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 a
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:

```haskell
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 some functions now:

```haskell
validateFounded :: Int -> Either FoundedError Founded
validateFounded n
  | n < 0     = Left $ NegativeYears n
  | n > 500   = Left $ TooManyYears n
  | otherwise = Right n


-- Tho, 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`. That Applicative and Monad must have the same behavior but you can't make a Monad for Validation that accumulates the errors like the Applicative does. Instead, it'll be identical to the Either monad.

Because that law must hold, we can't make a `Monad` that behaves like `Validation`'s `Applicative`, only like `Either`'s short-circuit behavior.

Exercise

Implement the Either Monad.

```haskell
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` typeclass has laws, just as the other typeclasses do. These laws exist, as with all the other typeclass 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:

```haskell
-- 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 `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]
```

Just 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 `>>=`:

```
(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 there, in which, for the right side of the equals sign, we had 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 `bind` 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 for getLine is evaluated first, then putStrLn is passed the input string that resulted 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 which gets incremented on each fmap or bind.

```haskell
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 = undefined :: CountMe (Int, String, Int)
  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 failed the laws because Functor did; in the

Applicative instance we were using 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 actually 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 instance before you've written any code. Incidentally, even if you fix the Functor and Applicative instances, there's no way to fix the Monad.

```haskell
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 **pure** setting the integer value to zero is fine for the purposes of the Applicative, but it violates the right identity law of Monad.

```
Prelude> CountMe 2 "blah" >>= return
CountMe 0 "blah"
```

So our "pure" is too opinionated. Still a valid Applicative and Functor, but what if `pure` didn't agree with the Monoid of the structure? The following will pass the Functor laws but it isn't a valid Applicative.

```haskell
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 just change the Monoid-of-structure to match the identity such that we have multiplication and the number one, it's a valid Applicative again.

```haskell
instance Applicative CountMe where
  pure = CountMe 1
  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!

```haskell
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 did, it works fine! It can be easy at times to accidentally write an invalid Monad that typechecks, 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 hadn't mattered much. Both concerned functions that looked like the usual `(a -> b)` arrangement, so composition "just worked" and that this was true was guaranteed by the laws of those typeclasses:

```
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:

```
mcomp :: Monad m => (b -> m c) -> (a -> m b) -> a -> m c
mcomp f g a = f (g a)
```

Well, that didn't work. What does? *Kleisli composition*, that's what. Don't sweat the strange name; it's not as weird as it sounds.

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:

```haskell
import Control.Monad

-- notice 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?

```haskell
(>=>) :: 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!

```haskell
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 used `return` composed with `read` to turn it into something that provides monadic structure after being bound over the output of `sayHi`. We needed the kleisli composition operator to stitch `sayHi` and `readM` together:

```
sayHi :: String -> IO String
readM :: Read a => String -> IO a

--  [1] [2] [3]        [4]        [5] [6]   [7]     [8] [9]
    (a -> m   b)      -> (b      -> m   c) -> a ->    m   c
String -> IO  String     String -> IO  a    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 which **sayHi** will produce.

5. The **IO** **readM** returns into. Note that **return/pure** produce **IO** values which perform no effects.

6. The **Int** that **readM** returns.

7. The original, initial **String** input **sayHi** expects so it knows how to greet the user and ask for their age.

8. The final combined **IO** action which 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** returned.

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.

```haskell
data Nope a =
  NopeDotJpg
```

```haskell
-- We're serious. Write it anyway.
```

2. ```haskell
   data PhhhbbtttEither b a =
       Left a
     | Right b
   ```

3. Write a Monad instance for Identity.

   ```haskell
   newtype Identity a = Identity a
       deriving (Eq, Ord, Show)
   ```

   ```haskell
   instance Functor Identity where
     fmap = undefined
   ```

   ```haskell
   instance Applicative Identity where
     pure = undefined
     (<*>) = undefined
   ```

   ```haskell
   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.

   ```haskell
   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 are fine, but it has to typecheck with types provided.

1. ```haskell
   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 Definition

1. A *monad* is a typeclass 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 which 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 which generates more structure after having been lifted over monadic structure. Contrast the function arguments to `fmap` and `(>>=)` in:

   ```
   fmap :: (a -> b) -> f a -> f b

   (>>=) :: m a -> (a -> m b) -> m b
   ```

   The significant difference is that the result 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:

```haskell
-- 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 just mean that we lifted 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 having bound over/out some a out of `m a`.

18.9   Follow-up resources

1. What a Monad is not
   https://wiki.haskell.org/What_a_Monad_is_not

2. Gabriel Gonzalez; How to desugar Haskell code
   http://www.haskellforall.com/2014/10/how-to-desugar-haskell-code.html

3. Stephen Diehl; What I wish I knew when Learning Haskell
   http://dev.stephendiehl.com/hask/#monads

4. Stephen Diehl; Monads Made Difficult
   http://www.stephendiehl.com/posts/monads.html

5. Brent Yorgey; Typeclassopedia
   https://wiki.haskell.org/Typeclassopedia

Chapter 19

Abstract structure applied

Monoid, Functor, Applicative, and Monad Gone Wild

> 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.

<div align="right">Jack Prelutsky</div>

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 for you to 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 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:

```haskell
{-# 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 this example 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.

Here we're using **foldr** and Monoid to concatenate connection parameters for connecting to the database:

```haskell
runDb :: SqlPersist (ResourceT IO) a -> IO a
runDb query = do
    params <- dbConnParams
    let connStr = foldr (\(k,v) t ->
        t <> (encodeUtf8 $ k <> "=" <> v <> " ")) "" params
    runResourceT . withPostgresqlConn connStr $ runSqlConn query
```

If you're interested in following up on this, 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 defaultConfig { keys =
      keys defaultConfig `mappend`
      \c -> fromList [
      ((0, xK_F6), lowerVolume 4 >> return ()),
      ((0, xK_F7), raiseVolume 4 >> return ())
      ]
  }
```

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 weak-head normal form. We'll explain this in more detail later in the book.

OK, so 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 based on that. 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 hadn't covered 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:

```
Prelude> import Data.Monoid
Prelude> let f = const (Sum 1)
Prelude> let 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 tried to apply them to and use the Monoid to combine the results:

```
Prelude> (f <> g) 9001
Sum {getSum = 3}
```

So this monoid instance allows 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 the Testing chapter. 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> M.fromList [('a', 1)] <> M.fromList [('a', 2)]
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 mappended functions, then combine 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 found especially demonstrative of why it's so handy.

Lifting over IO

Here we're taking a function that doesn't perform IO, `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 got 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`:

```haskell
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 web application which is also a Monad. It's a Monad because your "app context" will have a type parameter to describe what result was 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 just pattern match on the Maybe value, an AppHandler isn't something we can pattern match on. It's a Snap convention 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 suffice to say there are a few things floating around:

1. HTTP request which triggered the processing currently occurring.

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 which requires the reading in of configuration, 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 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 structure which handles all this work. It's like we're saying, "here's a function, apply it to a thing that resulted 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 example from Luke Hoersten's `hgrev` project. The example in the README is a bit dense, but 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 typeclass. 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 oper-
ator, `>>`) is useful when you're emitting effects. In this case, you've done
something with effects and want to discard any value that resulted.

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 further 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 but for CSV data:

```
parseRecord :: Record -> Parser a
```

```haskell
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 `=<<` as well:

```haskell
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 can make your code.

And now for something different

This next example is also using an applicative, but this is a bit different than the above examples. We'll spend more time explaining this one, as this pattern for writing utility functions is common:

```haskell
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. We start with
the operator for boolean disjunction, (||), which is an *or*:

```
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> let (<||>) = liftA2 (||)
```

And we'll make some trivial functions again for the purposes of demonstra-
tion:

```
Prelude> let f 9001 = True; f _ = False
Prelude> let 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` is 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** also by Michael Xavier. See the **network** library on Hackage for use and documentation:

```haskell
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, from the hangman game to the Monad chapter. The next example is a bit richer.

Binding over failure in initialization

Michael Xavier's Seraph is a process monitor and has a **main** function which is typical of more developed libraries and applications. The outermost Monad is just **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:

```haskell
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.[1]   There are some pieces we are not going to explain thoroughly; however, this is something you can actually 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
```

---

[1]The code in this example can be found here: `https://github.com/bitemyapp/shawty-prime/blob/master/app/Main.hs`

```
synopsis:           Initial project template from stack
description:        Please see README.md
homepage:           http://github.com/bitemyapp/shawty
license:            BSD3
license-file:       LICENSE
author:             Chris Allen
maintainer:         cma@bitemyapp.com
copyright:          2015, Chris Allen
category:           Web
build-type:         Simple
cabal-version:      >=1.10

executable shawty
  hs-source-dirs:      app
  main-is:             Main.hs
  ghc-options:         -threaded -rtsopts -with-rtsopts=-N
  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[2] and the Stack video tutorial[3] we worked on together. The code following from this point is in `Main.hs`.

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 typeclass. String literals are not ordinarily polymorphic; String is a concrete type. Using Overloaded-Strings 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 typeclass 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:

---

[3]Stack Github repo `https://github.com/commercialhaskell/stack`

[3]The video Stack mega-tutorial! The whole video is long, but covers a lot of abnormal use cases. Use the time stamps to jump to what you need to learn. `https://www.youtube.com/watch?v=sRonIB8ZStw&feature=youtu.be`

```haskell
class Num a where
  -- irrelevant bits elided
  fromInteger :: Integer -> a

class Num a => Fractional a where
  -- Elision again
  fromRational :: Rational -> a
```

Okay, and what about our literals?

```
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:

```haskell
{-# 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 it around a little. Note that, due to the monomorphism restriction, the following will work in the REPL but would not work if we loaded it from a source file (because it would default to a concrete type; we've seen this a couple times earlier in the book):

```
Prelude> :set -XOverloadedStrings
Prelude> let 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> import Data.ByteString (ByteString)
Prelude> let t = "blah" :: Text
Prelude> let 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` function to be invoked when the executable runs. We follow the OverloadedStrings extension with our imports:

```haskell
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 (. . .) as (. . .)" 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 that name 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:

```haskell
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:

```haskell
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) :: IO Int
  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 7 times, giving a list of 7 random letters or numbers:

```haskell
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 very important here. Redis can be convenient for some common use cases like caching, or when you want persistence without a lot of ceremony, as was 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 whose arguments are 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 created, 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)
            ]
```

```haskell
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 web-appy bits in the form of our application. We'll enumerate the application in chunks, but they're all in one `app` function:

```haskell
app :: R.Connection
    -> ScottyM ()
app rConn = do
-- [1]
  get "/" $ do
-- [2]
    uri <- param "uri"
--         [3]
```

1. Redis connection that should've been fired up before the web server started.

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.

   ```haskell
   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 param 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 ampersand to separate the key value pairs.

```
http://localhost:3000/?name=Blah&state=Texas
```

Now for the next chunk of the `app` function:

```haskell
    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 gave us a valid URI by using the `network-uri` library's `parseURI` function. We don't really care about the datatype it got 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 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 provided from a lazy Text value into a string Text value, then encoding 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 saved the short code for the URI. This gives the user a shortened URI to share.

7. Error response in case the user gave us a URI that wasn't valid.

The second handler handles requests to a shortened URI and returns the unshortened URL to follow:

```haskell
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 with curl/wget, you'd point your client at `http://localhost:3000/blah` to test it.

2. Same param 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 got Left so that the user knows what the error was, taking advantage of Redis having Showable errors to render it in the response.

6. Happy path.

7. Just because an error didn't happen doesn't mean the key was 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 we stored.

9. Going in the opposite direction we went in before — decoding the ByteString on the assumption 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. Our `main` function returns `IO ()` and acts as the entry point for our webserver 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:

```haskell
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 it after we've

covered monad transformers as well and 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 actually 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 first.

19.7   That's a wrap!

We hope this chapter gave you some idea of how Haskellers use the typeclasses 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 typeclasses with some interesting properties that rely on these four algebraic structures (monoid, functor, applicative, and monad), so we encourage

you to take some time to explore some of the uses we've demonstrated here. Consider going back to anything you didn't understand very well the first time you went through those chapters.

19.8   Follow-up resources

1. The case of the mysterious explosion in space; Bryan O'Sullivan; 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 typeclass 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 the chapter on folds, lists are certainly 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 typeclass 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 typeclass includes many operations. We're going to go through the full definition a little at a time. The definition in the library begins:

```
class Foldable t where
    {-# MINIMAL foldMap | foldr #-}
```

The `MINIMAL` annotation on the typeclass tells you that a minimally complete definition of the typeclass will define `foldMap` *or* `foldr` for a datatype. As it happens, `foldMap` and `foldr` can each be implemented in terms of the other, and the other operations included in the typeclass 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 typeclass 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 typeclass 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 very 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 while the Prelude as of GHCi 7.10 includes many changes related to the Foldable typeclass, not all of Foldable is in the Prelude. To follow along with the examples in the chapter, you will 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 :: Data.Monoid.Monoid m => t m -> m
  foldMap :: Data.Monoid.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 is requiring that you make the implicit Monoid visible in folding operations. Let's take a look at a very basic **foldr** operation and see how it compares to **fold** and **foldMap**:

```
Prelude> foldr (+) 0 [1..5]
15
```

The binary associative operation for that 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 the same 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 have a Monoid specified:

```
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, what we need to do to make **fold** work is specify a Monoid instance:

```
Prelude> fold [Sum 1, Sum 2, Sum 3, Sum 4, Sum 5]
Sum {getSum = 15}

-- or, less tediously:

Prelude> fold [1, 2, 3, 4, 5 :: Sum Integer]
Sum {getSum = 15}

Prelude> fold [1, 2, 3, 4, 5 :: Product Integer]
Product {getProduct = 120}
```

In some cases, the compiler can identify and use the standard Monoid for a type, without us being explicit:

```
Prelude> foldr (++) "" ["hello", " julie"]
"hello julie"
Prelude> fold ["hello", " julie"]
"hello julie"
```

The default Monoid instance for lists gives us what we need without having to specify it.

And now for something different

Let's turn our attention now to `foldMap`. Unlike `fold`, `foldMap` has a function as its first element. Unlike `foldr`, the first (function) argument of `foldMap` must explicitly map each element of the 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> foldMap First [Just 1, Nothing, Just 5]
First {getFirst = Just 1}
Prelude> foldMap Last [Just 1, Nothing, Just 5]
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> foldMap (*5) [1, 2, 3 :: Product Integer]
Product {getProduct = 750}
-- 5 * 10 * 15
750

Prelude> foldMap (*5) [1, 2, 3 :: Sum Integer]
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` in which 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> foldr (*) 3 [1, 2, 3 :: Sum Integer]
Sum {getSum = 18}
Prelude> foldr (*) 3 [1, 2, 3 :: Product Integer]
Product {getProduct = 18}
```

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> foldMap (*5) (Just 100) :: Product Integer
Product {getProduct = 500}

Prelude> foldMap (*5) (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> foldMap (*5) Nothing :: Sum Integer
Sum {getSum = 0}
Prelude> foldMap (*5) Nothing :: Product Integer
Product {getProduct = 1}
```

So, what we've seen so far is that Foldable is a way of generalizing catamorphisms — 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 typeclass 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 just so you have 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` — no need to make use of the Monoid instance for Identity since there is only one value, so we just apply the function.

It may seem strange to think of folding one value. When we've talked about catamorphisms previously, we've 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:

```
Main> foldr (*) 1 (Identity 5)
5
Main> foldl (*) 5 (Identity 5)
25
```

```
Main> foldMap (*5) (Identity 100) :: Product Integer
Product {getProduct = 500}
```

Note that using **foldMap** still relies on the Monoid instance for the *a* value, just as it did in the examples earlier.

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> foldMap (+1) Nothing :: Sum Integer
Sum {getSum = 0}
```

When the value is a Just value, though, we need to apply the folding function to the value and, again, dispose of the structure:

```
Prelude> foldr (+) 1 (Just 3)
4
Prelude> foldMap (+1) $ 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 it what Monoid you mean, it'll whinge 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 gave us `Sum 0` because that was the mempty or identity for Sum. Similarly with Nada and Product, we got `Product 1` because that was the identity for Product.

20.5 Some basic derived operations

The Foldable typeclass 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> map toList [Just 1, Just 2, Just 3]
[[1],[2],[3]]
Prelude> concatMap toList [Just 1, Just 2, Just 3]
[1,2,3]
Prelude> concatMap toList [Just 1, Just 2, Nothing]
[1,2]
Prelude> toList (1, 2)
[2]
-- it doesn't put the 1 in the list for the same reason
-- as fmap doesn't apply a function to the 1
```

```
-- | Test whether the structure is empty.
--   The default implementation is
--   optimized for structures that are similar
--   to cons-lists, because there
--   is no general way to do better.
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> fmap null [Just 1, Just 2, Nothing]
[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 arguments 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'.  The default
-- implementation is optimized for structures
-- that are similar to cons-lists, because there
-- is no general way to do better.
length :: t a -> Int
```

```
Prelude> length (1, 2)
1
Prelude> length [(1, 2), (3, 4), (5, 6)]
3
Prelude> fmap length [(1, 2), (3, 4), (5, 6)]
[1,1,1]

Prelude> fmap length Just [1, 2, 3]
1
-- the a of Just a in this case is a list, and
-- there is only one list

Prelude> fmap length [Just 1, Just 2, Just 3]
[1,1,1]
```

```
Prelude> fmap length [Just 1, Just 2, Nothing]
[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> fmap (elem 3) [Right 1, Right 2, Right 3]
[False,False,True]
```

```
-- | The largest element of a non-empty structure.
maximum :: forall a . Ord a => t a -> a
```

```
-- | The least element of a non-empty structure.
minimum :: forall a . 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> fmap maximum [Just 2, Just 10, Just 4]
[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> fmap minimum [Just 'j', Just 'u', Just 'l']
"jul"

Prelude> fmap minimum [Just 4, Just 3, Nothing]
[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:

```
-- | The 'sum' function computes the sum of the
-- numbers of a structure.
sum :: (Foldable t, Num a) => t a -> a

-- | The 'product' function computes the product
-- of the numbers of a structure.
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

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 **foldr** 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 -> a
   ```

5. ```
   maximum :: (Foldable t, Ord a) => t a -> 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 a
   ```

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 foldMap.

```
filterF :: (Applicative f, Foldable f, Monoid (f a)) =>
           (a -> Bool) -> f a -> f a
filterF = undefined
```

20.7   Answers

Filter function for Foldable types. It's a bit silly, but it does technically work. We need the Applicative so we have minimal structure to put a value into, then mappend with the rest of the structure.

```
filterF p = foldMap (\x -> if p x then pure x else mempty)
```

You have to specify the type for it to know what the output structure looks like.

```
Prelude> filterF (const False) (1, (Sum 1)) :: [Sum Int]
[]
Prelude> filterF (const True) (1, (Sum 1)) :: [Sum Int]
[Sum {getSum = 1}]

Prelude> filterF (const True) (1, (Sum 1)) :: Maybe (Sum Int)
Just (Sum {getSum = 1})
Prelude> filterF (const False) (1, (Sum 1)) :: Maybe (Sum Int)
Nothing
```

Running it on a list looks a bit like the following.

```
-- Prelude> filter' even [1, 2, 3, 4] :: [Int]
-- [2,4]

filter' even [1, 2, 3, 4] :: [Int]
```

20.8   Follow-up resources

1. Jakub Arnold, Foldable and Traversable
   http://blog.jakubarnold.cz/2014/07/30/foldable-and-traversable.
   html

Chapter 21

Traversable

O, Thou hast damnable
iteration; and art, indeed, able
to corrupt a saint.

Shakespeare

21.1   Traversable

Functor gives us a way to transform any values embedded in structure. Applicative, we saw, is a monoidal functor, and gives us a way to transform any values contained within a structure using a function that is also embedded in structure. This means that each application produces the effect of adding structure which is then applicatively combined. 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 didn't come until the release of GHC 7.10. However, it was available as part of the **base** library for much longer than that. Traversable depends on Applicative, and thus Functor, and is also superclassed by Foldable.

Traversable allows you to transform elements inside the 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 the applicative contexts along the way. This is easiest to see, perhaps, through liberal demonstration of examples, so let's get to it.

In this chapter, we will:

- explain the Traversable typeclass and its canonical functions;

- explore examples of Traversable in practical use;

- tidy up some code using this typeclass;

- and, of course, write some Traversable instances.

21.2   The Traversable typeclass definition

This is the typeclass definition as it appears in the library on Hackage (Data.Traversable):

```haskell
class (Functor t, Foldable t) => Traversable t where
    {-# MINIMAL traverse | sequenceA #-}

    -- | Map each element of a structure to an action,
    -- evaluate these actions from left to right, and
    -- collect the results. For a version that ignores
    -- the results see 'Data.Foldable.traverse_'.
    traverse :: Applicative f =>
               (a -> f b)
           -> t a
           -> f (t b)
    traverse f = sequenceA . fmap f

    -- | Evaluate each action in the structure from
    -- left to right, and collect the results.
    -- For a version that ignores the results see
    -- 'Data.Foldable.sequenceA_'.
    sequenceA :: Applicative f => t (f a) -> f (t a)
    sequenceA = traverse id
```

A minimal instance for this typeclass 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 flipping 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> fmap product [Just 1, Just 2, Nothing]
[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> sequenceA [Just 1, Just 2, Just 3]
Just [1,2,3]
Prelude> sequenceA [Just 1, Just 2, Nothing]
Nothing
Prelude> fmap sum $ sequenceA [Just 1, Just 2, Just 3]
Just 6
Prelude> fmap product (sequenceA [Just 3, Just 4, Nothing])
Nothing
```

In the first example, using **sequenceA** just 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 the 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 library has a function
called **catMaybes** that offers a different way of handling a list of Maybe
values:

```
Prelude> import Data.Maybe
Prelude> catMaybes [Just 1, Just 2, Just 3]
[1,2,3]
Prelude> catMaybes [Just 1, Just 2, Nothing]
```

```
[1,2]
Prelude> sum $ catMaybes [Just 1, Just 2, Just 3, Nothing]
6
Prelude> fmap sum $ sequenceA [Just 1, Just 2, Just 3, Nothing]
Nothing
```

Using `catMaybes` allows you to sum (or otherwise process) the 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 that and the types of `fmap` and `(=<<)` (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 similar to `flip bind`, that function is itself generating more structure. However, unlike `flip bind`, that structure can be of a different type than the structure we lifted over to apply the function. And at the end, it will flip the two structures around, as `sequenceA` did.

In fact, as we saw in the typeclass 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 just 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 being 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 fine.

Similarly, the type for `sequence` in GHC versions prior to 7.10 is just 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 just Applicative.

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 Traversable:

```
traverse :: Applicative f => (a -> f b) -> t a -> f (t b)

sequenceA :: Applicative f => t (f a) -> f (t a)
```

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> let f = undefined :: a -> Maybe b
```

---

[1]`http://hackage.haskell.org/package/vector`

```
Prelude> let 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 start looking 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 the Testing chapter with the Morse code example to look at a nice example of how to use `traverse`. Let's recall what we had done 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 (or, in the opposite conversion, 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 Data.Maybe> 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> let 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 here is for *any* Nothing values 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*). There is a **sequence** in Prelude and another, more generic, version in **Data.Traversable**:

```
Prelude> :t sequence
sequence :: Monad m => [m a] -> m [a]

-- more general, can be used with types
-- other than List
Prelude> import Data.Traversable as T
Prelude T> :t T.sequence
T.sequence :: (Traversable t, Monad m)
             => t (m a) -> m (t a)
```

To use this 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 for **fmap** to get applied to.

```
-- 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 the following:

```
*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 very 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 work in terms of types without worry about code that actually runs sometimes.

```haskell
-- This code is from Alex Petrov
-- who was kicking something around on Twitter.
-- 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 String
decodeFn :: String -> Either Err SomeObj
decodeFn = undefined

-- There's a query, that runs against DB and
-- returns array of strings
fetchFn :: Query -> IO [String]
fetchFn = undefined

-- there's some additional "context initializer",
-- that also has IO side-effects
makeIoOnlyObj :: [SomeObj] -> IO [(SomeObj, IoOnlyObj)]
makeIoOnlyObj = undefined

-- before
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 up this code. A few things made them suspicious:

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.

Here's what the pipeline function got pared down to:

```haskell
pipelineFn :: Query
           -> IO (Either Err [(SomeObj, IoOnlyObj)])
pipelineFn query = do
  a <- fetchFn query
  traverse makeIoOnlyObj (mapM decodeFn a)

-- Thanks to merijn on the IRC channel
-- for helping with this
```

We can make it pointfree if we want to:

```haskell
pipelineFn :: Query
           -> IO (Either Err [(SomeObj, IoOnlyObj)])
pipelineFn =
  (traverse makeIoOnlyObj . mapM decodeFn =<<) . fetchFn
```

And since `mapM` is just `traverse` with a slightly different type:

```haskell
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, pointfree style can help focus the attention on the functions, rather than 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 are changing 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 could poke and prod.

```
module HttpStuff where

import Data.ByteString.Lazy
import Network.Wreq

urls :: [String]
urls = [ "http://httpbin.com/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 typeclass. 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.

---

[2]`http://hackage.haskell.org/package/wreq`

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> let edgelordMap f t = runIdentity $ traverse (Identity . f) t
Prelude> :t edgelordMap
edgelordMap :: Traversable t => (a -> b) -> t a -> t b
Prelude> edgelordMap (+1) [1..5]
[2,3,4,5,6]
```

Using Const or Constant, we can recover a foldMappy looking Foldable as
well:

```
Prelude> import Data.Monoid
Prelude> import Data.Functor.Constant

Prelude> let xs = [1, 2, 3, 4, 5] :: [Sum Integer]
Prelude> traverse (Constant . (+1)) xs
Constant (Sum {getSum = 20})

Prelude> let foldMap' f t = getConstant $ traverse (Constant . f) t
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 intuitions for the rela-
tionships of these typeclasses 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 might see a progression:

```haskell
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 two-tuple or anonymous product:

```haskell
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

   ```haskell
   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 is generating the structure that appears on the "outside" of the traverse operation, there's no reason we shouldn't be able to float the function over the structure into the traversal itself.

2. Identity

   ```haskell
   traverse Identity = Identity
   ```

   This law says 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 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've been too surprising given what you've seen with `traverse`.

21.11   Quality Control

Great news! You can QuickCheck your Traversable instances as well, since they have laws. Conveniently, the *checkers* library we're been using already has the laws for us. You can add the following to a module and change the types alias to change what instances are being tested:

```haskell
type TI = []

main = do
  let trigger = undefined :: TI (Int, Int, [Int])
  quickBatch (traversable trigger)
```

Don't forget you'll need Arbitrary and EqProp instances! Why are you still staring at the book? Go, go do it.

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.

```haskell
newtype Identity a = Identity a
    deriving (Eq, Ord, Show)

instance Traversable Identity where
  traverse = undefined
```

Constant

```haskell
newtype Constant a b =
  Constant { getConstant :: a }
```

Maybe

```haskell
data Optional a =
    Nada
  | Yep a
```

List

```haskell
data List a =
    Nil
  | Cons a (List a)
```

Three

```haskell
data Three a b c =
  Three a b c
```

Three'

```haskell
data Three' a b =
  Three a b b
```

S

This'll suck.

```haskell
data S n a = S (n a) a

-- to make it easier, we'll give you the constraints.

instance Traversable n => Traversable (S n) where
  traverse = undefined
```

Instances for Tree

This might be hard. Write the following instances for Tree.

```haskell
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
   http://blog.jakubarnold.cz/2014/07/30/foldable-and-traversable.html

---

[3]Not a typo.

2. The Essence of the Iterator Pattern; Jeremy Gibbons and Bruno Oliveira.

3. Applicative Programming with Effects; Conor McBride and Ross Paterson.

Chapter 22

Reader

Functions waiting for input

> 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 typeclasses 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 Parsers 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, 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 the Reader Monad.

In this chapter, we will:

- examine the Functor, Applicative, and Monad 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 that this will help 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:

```haskell
import Control.Applicative

hurr = (*2)
durr = (+10)

m :: Integer -> Integer
m = hurr . durr
```

We know that the m function will take one argument because of the types of `hurr`, `durr`, and `(.)`. Note that if you do not specify the types and load it from a file, it will be monomorphic by default; if you wish to make m polymorphic, you may change its signature but you also need to specify a polymorphic type for the two functions it's built from. The rest of the chapter will wait while you verify these things.

When we apply m to an argument, `durr` will be applied to that argument first, and then the result of that will be passed as input to `hurr`. So far, nothing new.

We can also write that function composition this way:

```haskell
m' :: Integer -> Integer
m' = fmap hurr durr
```

We aren't accustomed to fmapping a function over another function, and you may be wondering what the functorial context here is. By "functorial context" we mean the structure that the function is being lifted *over* in order to apply 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 m', 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:

```haskell
fmap hurr durr x == (*2) ((+10) x)
-- when this x comes along, it's the
-- first necessary argument to (+10)
-- then the result for that is the
-- first necessary argument to (*2)
```

This is the Functor of functions. We're going to go into more detail about this soon.

For now, let's turn to another set of examples. Put these in the same file so `hurr` and `durr` are still in scope:

```haskell
m2 :: Integer -> Integer
m2 = (+) <$> hurr <*> durr

m3 :: Integer -> Integer
m3 = liftA2 (+) hurr durr
```

Now we're in an Applicative context. We've added 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 will work differently than fmapping did. This time, the argument will get passed to both `hurr` and `durr` in parallel, and the results will be added together.

`hurr` and `durr` are each waiting for an input. We can apply them both at once like this:

```haskell
Prelude> m2 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 (.) 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 (+) partially applied to the first argument which was doubled by (*2). There's a second argument, and that's what'll get added to the first argument that got 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
```

Okay, 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 we can assert a more concrete type for functions like `(<*>)` and see if the compiler agrees 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
Prelude> :t (<*>) :: (a -> a -> b) -> (a -> a) -> (a -> b)
(<*>) :: (a -> a -> b) -> (a -> a) -> (a -> b)
```

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 the `(*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 would share the same input and we want to apply some other function to the result of those to reach a final result. This happens more than you might think, and we saw an example of it back in the Abstract Structure Applied chapter:

```
module Web.Shipping.Utils ((<||>)) where

import Control.Applicative (liftA2)

(<||>) :: (a -> Bool) -> (a -> Bool) -> a -> Bool
(<||>) = liftA2 (||)
```

That is the same idea as `m3` above.

Finally, another example:

```
hurrDurr :: Integer -> Integer
hurrDurr = do
    a <- hurr
    b <- durr
    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, but we know it must be because we've used `do` syntax. We assign the variable *a* to the partially-applied function `hurr`, and *b* to `durr`. As soon as we receive an input, it will fill the empty slots in `hurr` and `durr`. 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.

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 just another way of abstracting out function application and 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

We'll be doing something here very similar to what you saw above, to give you practice and 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:

```haskell
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 `(.)` or `fmap`:

```
composed :: [Char] -> [Char]
composed = undefined

fmapped :: [Char] -> [Char]
fmapped = undefined
```

The output of those two should be identical: one string that is made all uppercase and reversed, like this:

```
Prelude> composed "Julie"
"EILUJ"
Prelude> fmapped "Chris"
"SIRHC"
```

Now we want to return the results of `cap` and `rev` both, as a tuple, like this:

```
Prelude> tupled "Julie"
("JULIE","eiluJ")
-- or
Prelude> tupled' "Julie"
("eiluJ","JULIE")
```

We will want to use an applicative here. The type will look 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 or Applicative instances.

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 Reader *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 bring up `: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 can be a little confusing, so we're going to unwind it until hopefully it's a bit more comfortable. First, let's see what we can accomplish with this:

```
Prelude> fmap (+1) (*2) 3
7
```

```
-- Rearranging a little bit syntactically
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 that seem so unusual here. Remember that `(->)` takes two arguments and therefore has 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 will be 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 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're going to remove the names of the functions
-- and the typeclass constraint as we can take it 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?

```haskell
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?

We can use the fact that we recognize this as function composition to make a slightly different instance for Reader:

```haskell
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 unpacked out of Reader.

3. Put the new function made from the composition back into Reader.

Without the Reader newtype, we drop steps 1 and 3 and have function composition.

Exercise

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 the 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 this:

```
String -> String -> String
```

really sucks when:

1. They aren't strictly *any* string value.

2. They aren't processed in an identical fashion. You don't handle ad-
   dresses the same as names.

So make the difference explicit.

We'll make two record types:

```haskell
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 merely some sample data to use. You can modify them as you'd like:

```haskell
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 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
```

The pattern of using Applicative in this manner is common, so there's an alternate way to do this using `liftA2`:

```
import Control.Applicative (liftA2)

-- with Reader, alternate
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

1. Write `liftA2` yourself. Think about it in terms of abstracting out the difference between `getDogR` and `getDogR'` if that helps.

```haskell
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.

```haskell
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 a pragma called **InstanceSigs**. It's an extension we need in order to assert a type for the typeclass methods. You ordinarily cannot assert type signatures in instances. The compiler already knows the type of the functions, so it's not usually necessary to assert the types in instances anyway. We did this for the sake of clarity, to make the Reader type explicit in our signatures.

```haskell
-- 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 return 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, we'll describe what you need to do and you write the code. If you unpack the type of Reader's apply 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 with apply, unlike fmap, also takes an argument of type r.

Make it so.

4. Rewrite the above example that uses Dog and Person to use your Reader datatype you just implemented the Applicative for. You'll need to change the types as well.

```
-- example type
getDogR :: Reader Person Dog
getDogR = undefined
```

22.7 The Monad of functions

Functions also have a Monad instance. You saw this in the beginning of this chapter, and you perhaps have some intuition now for how this must work. We're going to walk through a simplified demonstration of how it works before we get to the types and instance. Feel free to work through this section as quickly or slowly as you think appropriate to your own grasp of what we've presented so far.

Let's start by supposing that we could write a couple of simple functions like so:

```haskell
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 — increment the values inside our structure and also tell us the length of the value. We could write that like this:

```haskell
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 had. 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):

```haskell
barOne :: Foldable t => t a -> (t a, Int)
barOne r = (r, length r)
```

That gave us the reduction to one argument that we wanted but didn't increment the values in the list as our `foo` function does. We can add that this way:

```haskell
barPlus r = (foo r, length r)
```

But we can also do that more compactly by making **(foo r)** the first argument to `bar`:

```haskell
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:

```haskell
frooty' :: Num a => [a] -> ([a], Int)
frooty' = \r -> bar (foo r) r
```

Then we abstract this out so that it's not specific to these functions:

```haskell
fooBind m k = \r -> k (m r) r
```

In this very polymorphic version, the type signature will look like this:

```haskell
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 more clear by making some substitutions. We'll use the r to represent the argument that both of our functions are waiting on — the Reader-y part:

```haskell
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):

```haskell
(>>=)   :: 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
```

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:

```
-- with Reader Monad
getDogRM :: Person -> Dog
getDogRM = do
  name <- dogName
  addy <- address
  return $ Dog name addy
```

Exercise

1. Implement the Reader Monad.

   ```haskell
   -- 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 to make it work.

2. Rewrite the monadic `getDogRM` to use your Reader datatype.

22.8 Reader Monad by itself is kinda boring

It can't do anything the Applicative cannot.

Remember how we calculated the cardinality of types?

```haskell
{-# 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 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.

The idea to define the Monad instance of functions in terms of the Applicative using `flip` and `apply` is courtesy Joseph Tel Abrahamson.

22.9 You can change what comes below, but not above

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 with the withReaderT function which lets us start a new Reader context with a different argument being provided:

```
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 see the State monad where we can not only read in a value, but provide a new one which will change the value carried by the functions that called us, not only those we called.

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 really 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 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.

```haskell
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 uses **lookup** to find the value associated with a specified key in our zipped lists. For demonstration purposes, it's nice to have the outputs be predictable, so we recommend writing some that are concrete values, as well as one that can be applied to a variable:

```haskell
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
-- that first argument is a function
-- in this case, we want it to be addition
-- summed is just 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'll be using `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 Maybe value is a Nothing, it returns the default value instead:

```
*ReaderPractice> fromMaybe 0 xs
6
*ReaderPractice> fromMaybe 0 zs
0
```

Now we'll cobble together a `main` function, so that in one function 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 your `main` function):

```
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 that 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` function 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`.

4. apply `bolt` to `z'`.

Rewriting Shawty

Remember the URL shortener? Instead of manually passing the database connection `rConn` from the main function 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 them soon, but you should try to do the transformation mechanically. Research as necessary using a search engine.

22.12 Follow-up resources

1. Reader Monad; All About Monads
   `https://wiki.haskell.org/All_About_Monads`

2. Reader Monad; Programming with Monads; Real World Haskell

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;

- and 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 this maps onto the "state" that exists in computer memory.

In most imperative programming languages, this 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" which 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 which 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, but we will not be covering it in this book.

In Haskell, if we use the State type and its associated Monad (for convenience, not strictly necessary), we can have state which:

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 which 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 give an overview of some of the functions we'll be using here. We used the System.Random library back in the chapter where we built the hangman game, but we'll be using some different functions for this example. This is in broad strokes; it isn't meant to go into great detail about how these generators work.

---

[1]`https://hackage.haskell.org/package/random`

System.Random is designed to generate pseudorandom values. You can generate those values through providing a seed value or by using the system-initialised 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
   ```

   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 into 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 have a little demonstration of these:

```
Prelude> import System.Random
Prelude> mkStdGen 0
1 1
Prelude> :t mkStdGen 0
mkStdGen 0 :: StdGen
```

```
Prelude> let 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's deciding the values returned is pure; the type doesn't permit the performance of any effects to get spooky action. 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 feed the StdGen value from that tuple to `next` again. We can use `snd` to extract that StdGen value and pass it as an input to `next`:

```
Prelude> snd (next sg)
240084 40692
Prelude> let newSg = snd (next sg)
Prelude> :t newSg
newSg :: StdGen
Prelude> next newSg
(2092764894,1601120196 1655838864)
```

You'll keep getting the same results of `next` there, 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> randomR (0, 3) newSg :: (Int, StdGen)
(1,1601120196 1655838864)
Prelude> randomR (0, 3) newSg :: (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> let rx :: (Int, StdGen); rx = random (snd sg3)
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:

```haskell
newtype Reader r a =
  Reader { runReader :: r -> a }
```

Actually, we've seen several newtypes whose contents are a function, particularly with our Monoid newtypes (Sum, Product, etc.). Newtypes must have the same underlying representation as the type they wrap, as the newtype wrapper disappears at compile time. So 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 demonstrates an isomorphism:

```haskell
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 do not:

```haskell
-- Not an isomorphism, because it might not work.
(a -> Maybe b, b -> Maybe a)

-- Not an isomorphism for two reasons. You lose information
-- whenever there was more than one element in [a]. Also,
-- [a] -> a is partial because there might not be any
-- elements.
[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 of out the State type:

```
State :: (s -> (a, s)) -> State s a

runState :: State s a -> s -> (a, s)
```

State is a function that takes 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:

Note that `random` should look an awful lot like State to you here:

```
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

Now let us use this kit to generate die such as for a game:

```haskell
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.

```haskell
intToDie :: Int -> Die
intToDie n =
  case n of
    1 -> DieOne
    2 -> DieTwo
    3 -> DieThree
    4 -> DieFour
    5 -> DieFive
    6 -> DieSix
    -- Use this tactic _extremely_ 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]

---

[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.

Now we need to roll the dice:

```haskell
rollDieThreeTimes :: (Die, Die, Die)
rollDieThreeTimes = do
  -- this will produce the same results every
  -- time because it is free of effects.
  -- This is fine for this demonstration.
  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 dice roll 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 liability for such.

So, how can we improve our suboptimal code there. With State, of course!

```haskell
module RandomExample2 where

import Control.Applicative (liftA3)
import Control.Monad (replicateM)
import Control.Monad.Trans.State
import System.Random
```

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 install, so you should be good to go.

Using State will allow us to factor out the generation of a single Die:

```haskell
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:

```haskell
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:

```haskell
rollDie' :: State StdGen Die
rollDie' =
  intToDie <$> state (randomR (1, 6))
```

`State StdGen` had a final type argument of Int. We lifted `Int -> Die` over it and transformed that final type argument to Die. We'll exercise more brevity upfront in the next function:

```haskell
rollDieThreeTimes' :: State StdGen (Die, Die, Die)
rollDieThreeTimes' =
  liftA3 (,,) rollDie rollDie rollDie
```

Lifting the three-tuple constructor over three State actions that produce Die values when given an initial state to work with. How does this look 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 result. 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> take 6 $ evalState infiniteDie (mkStdGen 0)
[DieSix,DieSix,DieSix,DieSix,DieSix,DieSix]
```

We already know based on previous inputs that the first 3 values shouldn't be identical for a seed value of 0. So what happened? What happened is we repeated a single *die value* — we didn'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 wanted.

Keep on rolling

In the following example, we keep rolling a single die until we reach or exceed a sum of 20.

```haskell
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
```

Then seeing it in action:

```haskell
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:

```haskell
Prelude> :t randomIO
randomIO :: Random a => IO a
Prelude> (rollsToGetTwenty . mkStdGen) <$> randomIO
6
Prelude> (rollsToGetTwenty . mkStdGen) <$> randomIO
7
```

Under the hood, it's the same interface and State Monad driven mechanism, but it's mutating a global StdGen to walk the generator forward on each use. See the `random` library source code to see how this works.

Exercises

1. Refactor `rollsToGetTwenty` into having the limit be a function argument.

```
rollsToGetN :: Int -> StdGen -> Int
rollsToGetN = undefined
```

2. Change `rollsToGetN` to recording the series of die that occurred in addition to the count.

```
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 the libraries **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, just change them consistently throughout.

```
newtype Moi s a =
  Moi { runMoi :: s -> (a, s) }
```

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) = ???
```

---

[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.

```
Prelude> runMoi ((+1) <$> pure 0) 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 job in software with this one weird trick

Some companies will use FizzBuzz[4] to screen (not so much *test*) candidates applying to software positions. The problem statement goes:

> 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:

```haskell
fizzBuzz :: Integer -> String
fizzBuzz n | n `mod` 15 == 0 = "FizzBuzz"
           | n `mod` 5  == 0 = "Fizz"
           | n `mod` 3  == 0 = "Buzz"
           | otherwise       = show n

main :: IO ()
main =
  mapM_ (putStrLn . fizzBuzz) [1..100]
```

You will craft a fizzbuzz that makes gouts of blood come out of your interviewer's eye sockets using State. This is a suitable punishment for asking a software candidate to write this in person after presumably getting through a couple phone screens.

---

[4]http://c2.com/cgi/wiki?FizzBuzzTest

```haskell
import Control.Monad
import Control.Monad.State

fizzBuzz :: Integer -> String
fizzBuzz n | n `mod` 15 == 0 = "FizzBuzz"
           | n `mod` 5  == 0 = "Fizz"
           | n `mod` 3  == 0 = "Buzz"
           | otherwise       = show n


fizzbuzzList :: [Integer] -> [String]
fizzbuzzList list =
  execState (mapM_ addResult list) []

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]
```

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 pretty bad, 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 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 `[]` which has this property is the difference list[5] which has O(1) append.

---

[5]`https://github.com/spl/dlist`

```haskell
import Control.Monad
import Control.Monad.State
import qualified Data.DList as DL

fizzBuzz :: Integer -> String
fizzBuzz n | n `mod` 15 == 0 = "FizzBuzz"
           | n `mod` 5  == 0 = "Fizz"
           | n `mod` 3  == 0 = "Buzz"
           | otherwise       = show n


fizzbuzzList :: [Integer] -> [String]
fizzbuzzList list =
  let dlist = execState (mapM_ addResult list) DL.empty
  in DL.apply dlist [] -- convert back to normal list

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 was totally superfluous. That's good! It's not common you really *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 backwards.* 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

1. Construct a State where the state is also the value you return.

```haskell
get :: State s s
get = ???
```

Expected output

```haskell
Prelude> runState get "curryIsAmaze"
("curryIsAmaze","curryIsAmaze")
```

2. Construct a State where the resulting state is the argument provided and the value is defaulted to unit.

```haskell
put :: s -> State s ()
put s = ???

Prelude> runState (put "blah") "woot"
((),"blah")
```

3. Run the State with *s* and get the state that results.

```haskell
exec :: State s a -> s -> s
exec (State sa) = ???

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 which applies a function to create a new State.

```
modify :: (s -> s) -> State s ()
modify = undefined
```

Should behave like the following:

```
Prelude> runState (modify (+1)) 0
((),1)
Prelude> runState (modify (+1) >> modify (+1)) 0
((),2)
```

Note you don't need to compose them, you can just throw away the result because it returns unit for *a* anyway.

23.9   Follow-up resources

1. State Monad; All About Monads; Haskell Wiki
   `https://wiki.haskell.org/All_About_Monads`

2. State Monad; Haskell Wiki
   `https://wiki.haskell.org/State_Monad`

3. Understanding Monads; Haskell Wikibook

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 closely 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 also 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

```
Boy plays with dog.
```

might be thought to have an underlying representation such as

```
        S(entence)
      /    \
   Boy     plays (verb)
(subject)       \
            with (preposition)
               \
             dog (object)
```
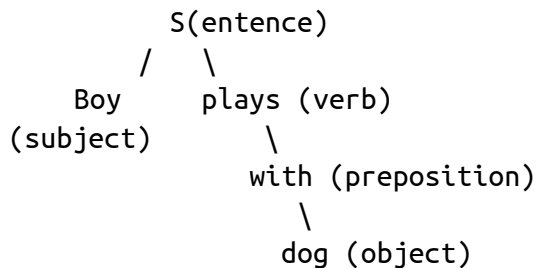
We are not here to become linguists, but parsing in computer science is related to the parsing of natural language sentences in some key ways. The core idea of parsing in programming is to accept serialized input in the form of a sequence of characters (textual data) or bytes (raw binary data) and turn 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.

---

[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.

If we wrote 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 can sometimes end up having a tree just 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 every sort of parser there is, or artisanally hand-craft 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. Just this topic could easily fill a book in itself (in fact, it has). The underlying types and typeclasses 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 in Haskell, 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, then you just started scanning with your eyes. Eventually, you learned how to read with sub-vocalization.

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 just going to use it in a state of somewhat ignorant bliss.

---

[2]We'll be using this version of trifecta
`http://hackage.haskell.org/package/trifecta-1.5.2`

OK, let's put up some code:

```haskell
module LearnParsers where

import Text.Trifecta

stop :: Parser a
stop = unexpected "stop"
```

`unexpected` is a means of throwing errors in parsers like Trifecta which are an instance of the Parsing typeclass. 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" just like that. However, if we did parse the word "bit" successfully and thus consumed 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 expression

```
char '1' >> stop
```

gets thrown away. Critically, whatever *effect* the `m a` action had upon the enclosing monadic context remains. The result value of the parse function gets thrown away, but the effect of "moving the cursor" remains.

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 which may or may not succeed. (A Nothing value means the parse failed.)

3. Return a tuple of the value you wanted and whatever's left of the string that you didn't consume to produce the 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 char-parsing function, we are simplifying here and this is not what the source code of any modern parsing library will look like:

```haskell
-- rudimentary char
char c =
  Parser $ \ s ->
    case s of
      (x:xs) -> if c == x
                  then Parser [(x, xs)]
                  else Parser []
      _ -> Parser []
```

We could encode the possibility of failure in that by adding Maybe but at this point, that 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 wanted to show you how the underlying function is the `s ->` embedded in the Parser data constructor.

Consider also the type of a classic Hutton-Meijer parser:

```haskell
-- 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> (runStateT $ put 2 >> get) 10021490234890
(2,2)
Prelude> (runStateT $ 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:

```haskell
-- 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 needed to declare the type of `testParse` in order to Show what we parsed 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 typeclasses. 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` function that will show us what we've got so far:

```haskell
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 labcoat and print the output using a dot-matrix printer. Some of you kids probably don't even know what a dot-matrix printer is.[3]

Run the **main** function and see what happens:

```
Prelude> main

stop:
Failure (interactive):1:1: error: unexpected
    stop
123<EOF>
^
```

We failed immediately before consuming any input in the above, so the caret in the error is at the beginning of our string value.

Next result:

---

[3]shakes fist at sky

```
one:
Success '1'
```

We parsed 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 dropped 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 parsed a single character successfully, then dropped it because we used `>>` to sequence it with `stop`. This means the cursor was one character forward due to the previous parser succeeding. Helpfully, Trifecta tells us where our parser failed.

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 parsed two characters individually. What if we we don't want to discard the first character we parsed and instead parse "12?" See the exercises below!

Intermission: Exercises

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 didn'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.

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 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> let 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> parseString (char 'b' >> char 'c') mempty "b"
Failure (interactive):1:2: error: unexpected
    EOF, expected: "c"
b<EOF>
 ^
Prelude> parseString (char 'b' >> char 'c') mempty "bc"
Success 'c'
Prelude> parseString (char 'b' >> char 'c') 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> parseString (char 'a') mempty "abcdef"
Success 'a'
Prelude> let stop = unexpected "stop pls"
Prelude> parseString (char 'a' >> stop) mempty "abcdef"
Failure (interactive):1:2: error: unexpected
    stop pls
abcdef<EOF>
 ^
Prelude> parseString (string "abc") mempty "abcdef"
Success "abc"
Prelude> parseString (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:

```haskell
{-# LANGUAGE OverloadedStrings #-}

module Text.Fractions where

import Control.Applicative
import Data.Ratio ((%))
import Text.Trifecta
```

We named 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 pragma in there later.

Now, on to parsing fractions! We'll start with some test inputs:

```haskell
badFraction = "1/0"
alsoBad = "10"
shouldWork = "1/2"
shouldAlsoWork = "2/1"
```

Then we'll write our actual parser:

```haskell
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
  print $ parseString parseFraction mempty shouldWork
  print $ parseString parseFraction mempty shouldAlsoWork
  print $ parseString parseFraction mempty alsoBad
  print $ parseString parseFraction mempty badFraction
```

Try not to worry about the `mempty` values; it 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 wanted of type *a* or an error string to let us know something went wrong. You can ignore the Delta thing — just 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 succeeded properly. The third failed because it couldn'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 did not result from the process of parsing; we know that because it is a `Success` data constructor. The final error resulted 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 which is tantamount to
Prelude> Success (1 % 0)
Success *** Exception: Ratio has zero denominator
```

This is sort of a problem because exceptions *end* our programs. Observe:

```
main :: IO ()
main = do
  print $ parseString parseFraction mempty badFraction
  print $ parseString parseFraction mempty shouldWork
  print $ parseString parseFraction mempty shouldAlsoWork
  print $ parseString parseFraction mempty alsoBad
```

We've put the expression that throws an exception in the first line this time, when we run it we get:

```
Prelude> main
Success *** Exception: Ratio has zero denominator
```

So, our program halted on the error. This is not great. The impulse of an experienced programmer is likely to be to want to "handle" the error. You *do not* want to attempt to catch or handle exceptions. Catching exceptions is okay, but this is a particular class of exceptions that means something is quite 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 not have exceptions or bottoms that aren't explicitly called out as a possibility in the types whenever possible.

If we were feeling keen to be a do-gooder, we could modify our program to handle the 0 denominator case and change it into a parse error:

```haskell
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 typeclass. Realistically, not all Monads have a proper implementation of `fail`, so it will be moved out into a `MonadFail` class eventually. For now, it suffices to know that it is our means of returning an error for the Parser type here.

Now for another run of our test inputs, but with our more cautious parser:

```haskell
testVirtuous :: IO ()
testVirtuous = do
  print $ parseString virtuousFraction mempty badFraction
  print $ parseString virtuousFraction mempty alsoBad
  print $ parseString virtuousFraction mempty shouldWork
  print $ parseString virtuousFraction mempty shouldAlsoWork
```

When we run this, we're going to 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 which explains the cause for the failure. Much better!

Intermission: Exercise

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's consumed all the input but there is no result to return from having done so. The result **Success ()** tells you the parse was successful and consumed 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 parsed 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 opted to use `trifecta`, as you've seen. 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[4] is split across two libraries, `parsers`[5] 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 typeclasses 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.

---

[5]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 too much about their source code. When you look at a library on Hackage, (unless you click to view the source code), what you're looking at is the API for that library.

[5]`http://hackage.haskell.org/package/parsers`

Typeclasses of Parsers

As we noted above, `trifecta` relies on the `parsers`[6] library for certain typeclasses. These typeclasses 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 typeclass `Parsing` has `Alternative` as a superclass. We'll talk more about Alternative in a bit. The Parsing typeclass provides for functionality needed to describe parsers independent of input type. A minimal complete instance of this typeclass 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 didn't consume 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 typeclass 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"
   ```

---

[6]http://hackage.haskell.org/package/parsers

We'll be seeing more of this one in upcoming sections.

3. The library also defines the typeclass `CharParsing`, which has `Parsing` as a superclass. This handles parsing 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 typeclasses 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 had a 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 had 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
  print $ parseString (some letter) mempty a
  print $ parseString integer mempty b
  print $ parseString parseNos mempty a
  print $ parseString parseNos mempty b
  print $ parseString (many parseNos) mempty c
  print $ parseString (some parseNos) mempty 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 typeclass:

```haskell
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 `some` and `many` are defined in `GHC.Base` because they come from this typeclass rather than being specific

to a particular parser or to the `parsers` library. Or really even a particular problem domain.

What if we wanted to require that each value be separated by 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|` is beginning a quasiquoted[7] section, using the quasiquoter named `r`. Note we had to enable the QuasiQuotes language extension to use this syntax. At time of writing `r` is defined in `raw-strings-qq` version 1.1 as follows:

---

[7]There's a rather nice wiki page and tutorial example at: `https://wiki.haskell.org/Quasiquotation`

```haskell
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"
```

The idea here is that this is a macro that lets us write arbitrary text inside of the block that begins with `[r|` and ends with `|]`. This specific quasiquoter exists to make it much nicer and more natural to write multiline strings without manual escaping. The quasiquoter is generating the following 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[8] 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:

---

[8]`https://wiki.haskell.org/Template_Haskell`

```
{-# 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 generated:

```
Prelude> :set -ddump-splices
Prelude> :l code/quasi.hs
[1 of 1] Compiling Quasimodo
code/quasi.hs:(8,12)-(12,2): Splicing expression
    "\n\
    \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 our `main` function to apply that to the eitherOr value:

```
main = do
  print $ parseString parseNos mempty eitherOr
```

Note that we lifted `Left` and `Right` over their arguments. This is because there is `Parser` structure between the eventual value running the parser will (potentially) obtain 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 just say that the type is specifically Trifecta's Parser type:

```
Prelude> let someLetter = some letter :: Parser [Char]
Prelude> let 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:

```
Prelude> 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 just need an **fmap** right?

```
-- same deal
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> let mynameParser = MyName <$> someLetter
Prelude> parseString mynameParser mempty "Chris"
Success (MyName "Chris")
```

Brill.

Back to our original code, which will actually spit out an error:

```
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 actually wanted 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
  print $ parseString parseNos mempty 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 started 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 which 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.

Intermission: Exercise

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 examples, we'll be using the INI[9] 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
```

---

[9]`https://en.wikipedia.org/wiki/INI_file`

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" which contains two settings: one named "host" with the value "wikipedia.org", another named "alias" with the value "claw".

We'll begin this example with our pragmas, module declaration, and imports:

```haskell
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE QuasiQuotes       #-}

module Data.Ini where

import Control.Applicative
import Data.ByteString (ByteString)
import Data.Char (isAlpha)
import Data.Map (Map)
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 combining 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 dang nifty. So, starting with a simple problem, let's create a test input for an INI header, a datatype, and then the parser for it:

```haskell
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 actually 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 passed to `parseBracketPair`:

```
Prelude> :t some letter
some letter :: CharParsing f => f [Char]
Prelude> :t Header <$> some letter
Header <$> some letter :: CharParsing f => f Header
Prelude> let slp = Header <$> some letter :: Parser Header
```

The first type is some parser that can understand characters which will produce a String value if it succeeds. The second type is the same, but 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 concrete Parser type from Trifecta where there had been 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 just to make the types more readable as well. Nothing too special here:

```haskell
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:

```haskell
name <- some letter
```

Then we parse and throw away the "=" used to separate keys and values:

```haskell
_ <- char '='
```

Then we parse one or more characters as long as they aren't a newline. This is so 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> parseString (some parseAssignment') mempty "key=value\nblah=123"
Success [("key","value")]
```

Pity. Can't parse the second assignment. But the first version that includes the `skipEOL` should work:

```
Prelude> let testS = "key=value\nblah=123"
Prelude> parseString (some parseAssignment) mempty testS
Success [("key","value"),("blah","123")]
Prelude> let daturrrr = "key=value\n\n\n\ntest=data"
Prelude> parseString (some parseAssignment) mempty daturrrr
Success [("key","value"),("test","data")]
```

We have to skip the one-or-more newline characters separating the first and second assignment 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 name and value together and re-embedding the result in the `Parser` type:

```
return (name, val)
```

Then for dealing with INI comments, where "dealing with" means skipping them in the parser and discarding 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, just as we did with comments above. This is also where we'll put that QuasiQuotes pragma to use, allowing us to make multiline strings nicer to write:

```haskell
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 getting on to the section parsing proper:

```haskell
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 defined 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 further more Maps of assignment names mapped to their values. We use `foldr` to aggregate the list of sections into a single Map value:

```haskell
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:

```haskell
parseByteString parseIni mempty sectionEx
```

and comparing it to the output of:

```haskell
parseByteString parseSection mempty sectionEx
```

that you saw above.

Now we'll put together our `main` function. Here we're interested in whether our parsers do what they should do rather than parsing an actual INI file, so we'll have the `main` function run some `hspec` tests. We'll use a helper function, `maybeSuccess`, as part of the tests:

```haskell
maybeSuccess :: Result a -> Maybe a
maybeSuccess (Success a) = Just a
maybeSuccess _ = Nothing
```

```haskell
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 "Can skip a comment before a 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. The parser then structures the stream of tokens into a tree, commonly called an "abstract syntax tree" or AST:

```
-- 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 purpose and class of grammar[10].

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"]
```

---

[10]https://en.wikipedia.org/wiki/Chomsky_hierarchy

```
Prelude> parseString (some integer) mempty "123"
Success [123]
Prelude> parseString (some integer) mempty "123456"
Success [123456]
```

The problem here is that if we wanted to recognize 123 and 456 as independent strings, we need some kind of separator. Now 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 space 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> parseString (token (some digit)) mempty "123 \n \n 456"
Success "123"
Prelude> parseString (token (some (token digit))) mempty "123 \n \n 456"
Success "123456"
```

```
Prelude> parseString (some decimal) mempty "123 \n \n 456"
Success [123]
Prelude> parseString (some (token decimal)) mempty "123 \n \n 456"
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 like `some integer` like this:

```
p' :: Parser [Integer]
p' = some $ do
  i <- token digit
  return (read [i])
```

And we can compare the output of that to the output of applying `token` to `digit`:

```
Prelude> parseString p' mempty "1\n2\n3"
Success [1,2,3]
Prelude> parseString (token (some digit)) mempty "1\n2\n3"
Success "1"
Prelude> parseString (some (token (some digit))) mempty "1\n2\n3"
Success ["1","2","3"]
```

You'll want to think carefully about what scope at which you're tokenizing as well:

```
Prelude> let 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 wanted that first example to work, we need to tokenize the parse of the first character, not the whole a-then-b parse:

```
Prelude> let 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 followed *a*. We can tokenize both character parsers though:

```
Prelude> let tknBoth = token (char 'a') >> token (char 'b')
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 thereof 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 typeclasses. Let's give it a whirl, shall we?

```haskell
{-# 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 typeclass-constrained polymorphism in our type signatures for flexibility. We'll write a **main** function that will run both **attoparsec** and **trifecta** versions for us so we can compare the outputs directly:

```haskell
main :: IO ()
main = do
  -- parseOnly is Attoparsec
  print $ parseOnly parseFraction badFraction
  print $ parseOnly parseFraction shouldWork
  print $ parseOnly parseFraction shouldAlsoWork
  print $ parseOnly parseFraction alsoBad

  -- parseString is Trifecta
  print $ parseString parseFraction mempty badFraction
  print $ parseString parseFraction mempty shouldWork
  print $ parseString parseFraction mempty shouldAlsoWork
  print $ parseString parseFraction mempty 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 which can then be run with various parsing libraries,

this doesn't free you of understanding the particularities of each if you want your parser to have predictable behavior. One of the areas particularly subject to this variation in behavior is backtracking, which we'll examine next.

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. Let's examine a few cases where this can cause different behavior between **trifecta**, **attoparsec**, and **parsec**:

```
Prelude> import Control.Applicative
Prelude> import Text.Trifecta
Prelude> import Text.Parsec (parse)
Prelude> let oP = string "ha" <|> string "hi" >> char '1'
Prelude> parse oP mempty "ha1"
Right '1'
Prelude> parse oP mempty "hi1"
Left (line 1, column 1):
unexpected "i"
expecting "ha"
```

Here we used **parsec** and it failed. This is because **parsec**'s Alternative instance providing **(<|>)** doesn't automatically backtrack for you. When it attempted the parse of "ha" before "hi," it had successfully parsed the character 'h', so the cursor was still on 'i'. The next parser didn't have a single correct input, so it attributed the failure of the parse to **string "ha"**.

What happens if we try the same, but with **attoparsec** and **trifecta**?

```
Prelude> :set -XOverloadedStrings
Prelude> import Data.Attoparsec.ByteString (parseOnly)
Prelude> parseOnly oP "hi1"
Right '1'
Prelude> parseString oP mempty "hi1"
```

```
Success '1'
```

These succeed because the Alternative instances providing (**<|>**) defined for these libraries automatically backtrack. The result of this is that when the first string rule fails, the cursor is rolled back to where it was when the parse of alternatives started, rather than leaving it where it was after any partial success. Making parses more atomic and easier to reason about is part of the motivation for tokenized parsing.

So how do we fix this? Add backtracking manually — this is what **try** was for. The following should have the same behavior in all three parsers:

```
Prelude> let s = string
Prelude> let p = try (s "ha") <|> s "hi" >> char '1'
Prelude> parse p mempty ("hi1" :: String)
Right '1'
Prelude> parseString p mempty "hi1"
Success '1'
Prelude> parseOnly p "hi1"
Right '1'
```

One potentially confusing niggle to this is that the smallest unit of stream consumption is by default going to be a character, unless you've tokenized the stream already. As a result, the following will work with **parsec** despite what you may have expected after seeing the above:

```
Prelude> let orP = char 'a' <|> char 'b' >> char '1'
Prelude> parse orP mempty "b1"
Right '1'
```

trifecta and **attoparsec** match each other in backtracking behavior so far as we're aware, but there might be exceptions so if you use this trick for reusing parsers, you'll want to be careful. This is fairly convenient as **trifecta** replaces most of the use-cases you'd want **parsec** for, with **attoparsec** serving for performance-sensitive production use-cases. Simon

Hengel's **attoparsec-parsec**[11] library is worth looking at if you'd like to understand the differences in more detail.

Keep your the scope of your backtracking small and focused. You'll want to eye token parsers carefully as they are a common source of backtracking behavior that may not have occurred to you in the heat of the moment.

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 difficult to understand 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 extremely 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, then you might be able to write a proper program. One of the mistakes programmers make in writing programs handling 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 was parsed, but from there, you might expect that AST or representation to take a particular *form*. This means we want to narrow the cardinality and get *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:

---

[11]http://hackage.haskell.org/package/attoparsec-parsec

```
      Text -> Structure  -> Meaning
--   parse -> unmarshal

   Meaning  -> Structure -> Text
-- marshal  -> serialize
```

There isn't only one way to accomplish this, but we'll show you a commonly used library and how it has this two-stage pipeline in the API.

Marshalling and unmarshalling JSON data

aeson is presently the most popular JSON[12] library in Haskell. One of the things that'll confuse programmers coming to Haskell from Python, Ruby, Clojure, JavaScript, or similar languages is that there's usually no unmarshal/marshall step. 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

sectionJson :: ByteString
sectionJson = [r|
{ "section":  {"host": "wikipedia.org"},
  "whatisit": {"red": "intoothandclaw"}
}
|]
```

---

[12]https://en.wikipedia.org/wiki/JSON

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 like the following:

Provided a strict ByteString when a lazy one was 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 provided; the "expected type" is what the types wanted. Either we used the wrong code (so expected type needs to change), or we provided the wrong values (actual type, our types/values, need to change). You can reproduce this error by making the following mistake in the marshalling module:

```
-- Change the import of the ByteString type constructor from:
import Data.ByteString.Lazy (ByteString)

-- Into:
import Data.ByteString (ByteString)
```

Provided a lazy ByteString when a strict one was expected

```haskell
{-# 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 *nota bene*, where the internal modules are mentioned. The key is to remember 'actual type' means "your code", 'expected type' means "what they expected," 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 datatype JSON AST — "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:

```haskell
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE QuasiQuotes       #-}

module Marshalling where

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 = decode sectionJson :: Maybe TestData
  print d
```

This will in fact net you a type error complaining about there not being an instance of FromJSON for TestData. Which is true! GHC has no idea how to unmarshal 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 -XQuasiQuotes
Prelude> decode "{\"blue\": \"123\"}" :: Maybe Color
Just (Blue "123")
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 is accepting the Value

type and ToJSON instances generate the Value type, closing the following loop:

```haskell
-- FromJSON
ByteString -> Value -> yourType
-- parse -> marshal

-- ToJSON
  yourType -> Value -> ByteString
-- marshal -> serialize
```

The definition of Value at time of writing is the following:

```haskell
-- | 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 unmarshal something that could be a Number or a String?

```haskell
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 a IEEE-754[13] 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 Chapter4, Basic Datatypes). So we need to convert from a Scientific to an Integer:

```haskell
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"
```

Now let's give it a whirl:

```
Prelude> decode "123" :: Maybe NumberOrString
Nothing
```

??? WHAT HAPPEN ???

We can use `eitherDecode` to get a *slightly* more helpful type error:

```
Prelude> eitherDecode "123" :: Either String NumberOrString
Left "Failed reading: satisfy"
```

---

[13]https://en.wikipedia.org/wiki/IEEE_floating_point

Not super helpful (it's **attoparsec** under the hood, after all), but it tells us it was parsing rather than marshalling that failed. Turns out, in the JSON standard, you'll find that only JSON arrays and objects are allowed to be at the top level of an input stream. There's a quick-n-dirty fix for this:

```
Prelude> type Result = Either String [NumberOrString]
Prelude> eitherDecode "[123]" :: Result
Right [Numba 123]
Prelude> eitherDecode "[\"123\"]" :: Result
Right [Stringy "123"]
```

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> parseString parseSemVer mempty "2.1.1"
SemVer 2 1 1 [] []
Prelude> parseString parseSemVer mempty "1.0.0-x.7.z.92"
SemVer 1 0 0 [NOSS "x", NOSI 7, NOSS "z", NOSI 92] []
Prelude> SemVer 2 1 1 [] [] > SemVer 2 1 0 [] []
True
```

2. Write a parser for positive integer values.

```
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 to do this.

```
Prelude> parseString base10Integer' mempty "-123abc"
Success (-123)
```

4. Write a parser for US/Canada phone numbers with varying formats.

```haskell
type NumberingPlanArea = Int -- aka area code
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". 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?
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 which 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.

```haskell
import Data.Word

data IPAddress =
  IPAddress Word32
  deriving (Eq, Ord, Show)
```

A 32-bit word is a 32-bit unsigned int. Lowest value is 0 rather than 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.

```haskell
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 abbrevations. See this Q&A thread[14] about IPv6 abbreviations for more.

Ensure you can parse abbreviated variations of the earlier examples like:

```
FE80::0202:B3FF:FE1E:8329
2001:DB8::8:800:200C:417A
```

8. Remove the derived Show instances from the IPAddress/IPAddress6 types, 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 IPAddress and IPAddress6.

10. Write a parser for the DOT language[15] Graphviz uses to express graphs in plain-text.

   We suggest you look at the AST datatype in Haphviz[16] for ideas on how to represent the graph in a Haskell datatype. If you're feeling especially robust, you can try using fgl[17].

---

[14]http://answers.google.com/answers/threadview/id/770645.html
[15]http://www.graphviz.org/doc/info/lang.html
[16]http://hackage.haskell.org/package/haphviz
[17]http://hackage.haskell.org/package/fgl

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. Good examples of this are things like 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 non-linear 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 unmarshaling. Cf. *serialization* and *deserialization.*

4. A *token(izer)* converts text, usually a stream of characters, into more meaningful or "chunkier" structures such as words, sentences, or symbols. The `lines` and `words` functions you've used earlier in this book are like very unsophisticated tokenizers.

5. *Lexer* — see tokenizer.

24.13 Follow-up resources

1. Parsec try a-or-b considered harmful; Edward Z. Yang

2. Code case study: parsing a binary data format; Real World Haskell

3. The Parsec parsing library; Real World Haskell

4. An introduction to parsing text in Haskell with Parsec; James Wilson;
   `http://unbui.lt/#!/post/haskell-parsec-basics`

5. Parsing CSS with Parsec; Jakub Arnold;
   `http://blog.jakubarnold.cz/2014/08/10/parsing-css-with-parsec.html`

6. Parsec: A practical parser library; Daan Leijen, Erik Meijer;
   `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.24.5200`

7. How to Replace Failure by a List of Successes; Philip Wadler;
   `http://dl.acm.org/citation.cfm?id=5288`

8. How to Replace Failure by a Heap of Successes; Edward Kmett

9. Two kinds of backtracking; Samuel Gélineau (gelisam);
   `http://gelisam.blogspot.ca/2015/09/two-kinds-of-backtracking.html`

10. LL and LR in Context: Why Parsing Tools Are Hard; Josh Haberman
    `http://blog.reverberate.org/2013/09/ll-and-lr-in-context-why-parsing-tools.html`

11. Parsing Techniques, a practical guide; second edition; Grune & Jacobs

12. Parsing JSON with Aeson; School of Haskell

13. aeson; 24 days of Hackage; Oliver Charles;
    `https://ocharles.org.uk/blog/posts/2012-12-07-24-days-of-hackage-aeson.html`

Chapter 25

Composing types

E pluribus monad

> 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 *magick*, so we want to approach them from both angles and demonstrate that they are both comprehensible via their types and quite 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 many times in "real code" 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 which 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 when 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 peoples' 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 `(.)`.

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:

```haskell
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 like 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, as newtypes, as we 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 just a means of wrapping one extra layer of (monadic) structure around a 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 it, 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* represent *type constructors*, not term-level functions:

```haskell
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:

```haskell
Compose :: (* -> *) -> (* -> *) -> * -> *
```

Does that remind you of anything?

```haskell
(.) :: (b -> c) -> (a -> b) -> a -> c
```

So, what does that look like in practice? Something like this:

```haskell
Prelude> Compose [Just 1, Nothing]
Compose {getCompose = [Just 1,Nothing]}

Prelude> :t Compose [Just (1 :: Int), Nothing]
Compose [Just (1 :: Int), Nothing] :: Compose [] Maybe Int
```

Given the above value, the type variables get bound accordingly:

```haskell
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 little functors sittin' in a tree, L-I-F-T-I-N-G

Let's start with composing functors, using the types we've seen just 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 really do anything interesting but captures the essence of what Functors are 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 ask 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 Functors themselves. We need to be able to jump over both those layers in order to apply to the value that's ultimately inside. We have to fmap twice to get to that value inside because of the layered structures.

To return to the example we used above, we have this type:

```haskell
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:

```haskell
Prelude> Compose [Just 1, Nothing]
Compose {getCompose = [Just 1,Nothing]}
Prelude> fmap (+1) (Compose [Just 1, Nothing])
Compose {getCompose = [Just 2,Nothing]}
```

We can generalize this to different amounts of structure, such as with one less bit of structure. You may remember this from a previous chapter:

```haskell
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:

```haskell
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 it happens, just as with the anonymous product (,) and the anonymous sum Either, the Compose type allows us to express arbitrarily nested types:

```haskell
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 just 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 indeed compose two types that have Applicative instances and get a new Applicative instance. But you're going to write it.

GOTCHA! Exercise time

```
-- instance types provided 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 full power of the Monad. This is one of those benefits. To compose Applicatives, you don't need to do the legwork that monads require 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:

```haskell
{-# 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:

```haskell
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:

```haskell
(Monad f, Monad g) => f (g a) -> (a -> f (g b)) -> f (g b)
```

Or formulated differently:

```haskell
(Monad f, Monad g) => f (g (f (g a))) -> f (g a)
```

And this is not possible. There's not a 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. You can also read Composing monads[1] by Mark P. Jones and Luc Duponcheel to see why it's not possible.

_____

[1] http://web.cecs.pdx.edu/~mpj/pubs/RR-1004.pdf

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 that after a little break for some exercises.

25.6 Intermission: Exercises

Compose Foldable

Write the Foldable instance for Compose. 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
```

Compose Traversable

Write the Traversable instance for Compose.

```
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.

```haskell
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 just one.
Write Bifunctor instances for the following types:

1. The less you think, the easier it'll be.

   ```haskell
   data Deux a b = Deux a b
   ```

2. ```haskell
   data Const a b = Const a
   ```

3. ```haskell
   data Drei a b c = Drei a b c
   ```

4. ```haskell
   data SuperDrei a b c = SuperDrei a b
   ```

5. ```haskell
   data SemiDrei a b c = SemiDrei a
   ```

6. ```haskell
   data Quadriceps a b c d =
       Quadzzz a b c d
   ```

7. ```haskell
   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 can'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 (2, 3, 4...) types that each have a Monad instance by wrapping around existing monads that provide each bit of wanted functionality.

The types are tricky here, so we're going to be walking through writing monad transformers very 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 (>>=) which 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 Readers (app-specific data vs. what the framework provides by default), although usually there's a way to just add 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?

```
IO (Reader String [a])

-- where the Monad instances involved
-- are that of IO, Reader, and []
```

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; we can 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

Just 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:

```haskell
-- Plain old Identity. 'a' can be something with
-- more structure, but it's not required and
-- Identity won't know anything about it.
newtype Identity a =
  Identity { runIdentity :: a }
  deriving (Eq, Show)

-- The identity monad transformer, serving only to
-- to specify that additional structure should exist.
newtype IdentityT f a =
  IdentityT { runIdentityT :: f a }
  deriving (Eq, Show)
```

What changed here is that we added an extra type argument.

Then we want Functor instances for both Identity and IdentityT:

```haskell
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 here 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:

```haskell
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:

```haskell
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 have to really 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   >>= runIdentityT . f
--        [8]        [4] [5]    [7]        [6]
```

1. First we pattern-match or unpack the `m a` value of `IdentityT m a` via the data constructor. Doing this 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. This is the function we're binding over `IdentityT m a`. It has the following type:

   ```
   (a -> IdentityT m b)
   ```

4. Here `ma` is the same one we unpacked out of the `IdentityT` data constructor and has the type `m a`. Removed from its IdentityT context, this is now the `m a` that this 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 asked 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 typeclass 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* which was 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 =>` 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> let f :: (a -> IdentityT m b); f = undefined
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` which the expression `ma >>= runIdentityT . f` returns and *repack* it in `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> IdentityT [1, 2, 3] >>= (return . (+1))
IdentityT {runIdentityT = [2,3,4]}
```

Implementing the bind, step by step

Now 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** bind-
ings 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 just
get a **let** binding in place and see it load, even if the code doesn't actually
work:

```haskell
(>>=) :: 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 that. 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 typecheck (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*:

```haskell
(>>=) :: 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 just asserted for *aimb* is impossible; we've just said it could be every 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* actually 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 is
-- basically our having lifted over it
-- Problem is, we >>='d

(a -> IdentityT m b)

-- over

m a

-- and got

m (IdentityT m b)
```

It doesn't typecheck 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've 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 having separate `fmap` and `join` instead of using `(>>=)` because it makes the step-wise 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:

```
-- Trying to 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 *a*1 here is the *a* we had assigned to *aimb*, but it's telling us that our actual type is not what we asserted 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` which 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 removed the type declaration for `aimb` and also changed 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 to make it typecheck:

```
-- 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 should we improve this line?

```
IdentityT $ join (fmap runIdentityT (fmap f ma))
```

Well, one of the Functor laws tells us something about fmapping 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)
```

Now it seems suspicious that we're fmapping and also using `join` on the result of having fmapped the two functions we composed. Isn't `join` plus `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 work still! 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 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 actually 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/Monad at our disposal isn't enough to make that new Monad instance. So what was novel in the following code?

```haskell
(>>=) :: IdentityT m a
      -> (a -> IdentityT m b)
      -> IdentityT m b
(IdentityT ma) >>= f =
  IdentityT $ ma >>= runIdentityT . f
```

Well, it wasn't the pattern match on `IdentityT`; we get that from the Functor anyway:

```haskell
-- Not this
(IdentityT ma) ...
```

It wasn't the ability to `(>>=)` functions over the `ma` value of type ma, we get that from the Monad constraint on m anyway.

```haskell
-- Not this
... ma >>= ...
```

We needed to know one of the types concretely so that we could use `runIdentityT` (essentially fmapping a *fold* of the IdentityT structure) and then repack the value in IdentityT:

```haskell
-- We needed to know IdentityT
-- concretely to be able to do this
IdentityT .. runIdentityT ...
```

As you'll recall, until we used `runIdentityT` we couldn't get the types to fit because `IdentityT` was 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 just 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 we have concrete information for.

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 just 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 be concrete; f can remain polymorphic so long as it has a Monad instance, so we only 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 IdentityT's Monad:

```
(IdentityT ma) >>= f =
  let aimb :: m (IdentityT m b)
      aimb = fmap f ma
```

We have *something* that'll typecheck, but it's not quite in the shape we would like. Of course, the underlying type once we throw away the IdentityT data constructor is `m (m b)` which'll 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 lifted the record accessor for IdentityT over the m! Since IdentityT is so simple, the record accessor is sufficient to "fold away" 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 is 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 to begin with.

25.10 Answers

Compose Applicative

First, a hint. You'll want to apply applicatively for one layer of structure. The trick is realizing you want to apply applicatively twice but somehow need to "get over" the extra structure in order to do so.

Then for the answer we're going to provide a *very* deconstructed — decomposed, if you will — implementation of the Applicative instance for `Compose`.

JUST SO WE'RE CLEAR: THIS IS NOT THE WAY YOU MUST WRITE THE INSTANCE. WE BROKE IT DOWN THIS FAR IN ORDER TO SHOW YOU THE STEPS INVOLVED. HOWEVER, FROM THIS, YOU SHOULD BE ABLE TO WRITE A MORE SENSIBLE-LOOKING INSTANCE:

```
{-# LANGUAGE InstanceSigs        #-}
{-# LANGUAGE ScopedTypeVariables #-}

-- assume type definition and
-- Functor instance are present

instance forall f g .
        (Applicative f, Applicative g) =>
         Applicative (Compose f g) where
    pure x = Compose (pure (pure x))

    (<*>) :: forall a b .
           Compose f g (a -> b)
        -> Compose f g a -- f (g a)
        -> Compose f g b -- f (g b)
    Compose f <*> Compose x =
      let liftApply :: f (g (a -> b))
                    -> f (g a -> g b)
          liftApply func = (<*>) <$> func

          apF :: f (g a -> g b)
          apF = liftApply f

          apApF :: f (g a) -> f (g b)
          apApF = (<*>) apF
      in Compose (apApF x)
```

The `ScopedTypeVariables` pragma is so we can talk about type variables more than once within the same scope. Ordinarily, a type is instantiated anew in every type signature. Here's an example of something that seems like it should work but doesn't:

```
module STV where

fapp :: (a -> b)
     -> a
     -> b
fapp f x =
  let result :: b
      result = f x
  in  result
```

The issue is that the second mention of *b* in the type signature of `result` creates a whole new polymorphic type variable *b* which isn't known to be the same thing as the *b* instantiated by the type signature of `fapp`.

We can fix this by adding the pragma and then explicitly instantiating *a* and *b* as scoped type variables:

```
{-# LANGUAGE ScopedTypeVariables #-}
```

```
module STV where

fapp :: forall a b . (a -> b)
     -> a
     -> b
fapp f x =
  let result :: b
      result = f x
  in  result
```

Chapter 26

Monad transformers

Stack 'em up

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 o' magick happen. Monad transformers are important in a lot of everyday Haskell code, though, so we want to dive deeper and make sure we have 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 at various times, though, the Maybe Monad can be extremely useful and so it is that the tranformer 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, because this 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:

```haskell
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:

```haskell
-- Remember the Functor for Compose?

instance (Functor f, Functor g) =>
         Functor (Compose f g) where
  fmap f (Compose fga) =
    Compose $ (fmap . fmap) f fga

-- compare 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. If you were stuck on it
and frustrated by it, there is a hint in the answers section of the previous
chapter and you may want to look at that now, before reading this section.
Or just jump right in — we just wanted to warn you, as this section may
ruin that exercise for you if you hadn't done it yet.

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 f (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? We did offer a demonstration in the answers of why you have to use the strange-looking (<*>) <$> configuration.

The idea here is that we have to lift an Applicative "apply" over the outer structure *f* to get the g (a -> b) into g a -> g b so that the Applicative instance for *f* can be leveraged. We can stretch this idea a bit and use concrete types:

```haskell
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 actual Applicative instance for such a hypothetical type would look like:

```haskell
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:

```haskell
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:

```haskell
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          ]
            -- y :: a
            -- f :: a -> MaybeT m b
            -- f y :: MaybeT m b
            -- runMaybeT (f y) :: m b
            Just y  -> runMaybeT (f y)
--              [7]        [8]
```

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 b` in order to typecheck 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 unbundled 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 out of the *m* structure which is unknown.

5. Since using `<-` to bind `Maybe a` out of `m (Maybe a)` left us with a Maybe value, we do a plain old case expression on the Maybe value.

6. If we get Nothing, we kick Nothing back out, but we have to re-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** (**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 b` value out of the MaybeT since the MaybeT constructor is already wrapped around the whole **do**-block, then we're done.

Don't be afraid to get a pen and paper and work all that out until you really 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:

```haskell
newtype EitherT e m a =
  EitherT { runEitherT :: m (Either e a) }
```

EitherT Exercises

1. Write the Functor instance for EitherT:

   ```haskell
   instance Functor m => Functor (EitherT e m) where
     fmap = undefined
   ```

2. Write the Applicative instance for EitherT:

   ```haskell
   instance Applicative m => Applicative (EitherT e m) where
     pure = undefined

     f <*> a = undefined
   ```

3. Write the Monad instance for EitherT:

   ```haskell
   instance Monad m => Monad (EitherT e m) where
     return = pure

     v >>= f = undefined
   ```

4. Write the `swapEitherT` helper function for EitherT.

   ```haskell
   -- transformer version of swapEither.
   swapEitherT :: (Functor m) => EitherT e m a -> EitherT a m e
   swapEitherT = undefined
   ```

   Hint: write `swapEither` first, then `swapEitherT` in terms of the former.

5. Write the transformer variant of the `either` catamorphism.

```haskell
eitherT :: Monad m =>
           (a -> m c)
        -> (b -> m c)
        -> EitherT a m b
        -> m c
eitherT = undefined
```

26.4   ReaderT

We mentioned back in the Reader chapter that, actually, you more often see ReaderT than Reader in common Haskell use. ReaderT is one of the most commonly used transformers in conventional Haskell applications. It is just like Reader, except in the transformer variant we're generating additional structure in the return type of the function:

```haskell
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 senses, 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*!

```haskell
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 the 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-matched 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've bound that value to the name a as a mnemonic to remember the type.

4. Applying f, which has 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 had 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) }
```

StateT Exercises

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. Just 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 chapter on nonstrictness.

1. You'll have to do the Functor and Applicative instances first, because there aren't 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 re-use 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*. To learn more, see this Stack Overflow question[1] about this issue. Also see this Github issue[2] on the NICTA Course Github repository. **Beware!** The NICTA course issue gives away the answer. In essence, the issue is that without Monad, you're just 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
   ```

---

[1] Is it possible to implement '(Applicative m) => Applicative (StateT s m)'? `http://stackoverflow.com/questions/18673525/is-it-possible-to-implement-applicative-m-applicative-statet-s-m`

[2] `https://github.com/NICTA/course/issues/134`

ReaderT, WriterT, StateT

We'd like to point something out about these three types:

```haskell
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:

```haskell
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 it. 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. Now you know why you don't *need* Writer; we'll talk 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 that 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 remember our discussion about the similarities between parsers and State in the Parsers chapter. 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 necessary 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 solving, 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.[3]

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 actually 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 wanted a stricter Writer than the Strict Writer 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 nonstrictness.

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. 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 really worth it for the reasons listed below.

2. It's not very fast.

3. Streaming libraries like `pipes`[4] and `conduit`[5] do it better for most use-cases.

---

[3]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`

[4]`http://hackage.haskell.org/package/pipes`

[5]`http://hackage.haskell.org/package/conduit`

Prior art for "ListT done right" also includes Amb/AmbT[6] by Conal Elliott, although you will probably 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 Recovering the ordinary type from the transformer

If you have a transformer variant of a type and want to use it as if it was the non-transformer version, you need some *m* structure that doesn't really do anything. Have we seen anything like that? What about `Identity`?

```
Prelude> runMaybeT $ (+1) <$> MaybeT (Identity (Just 1))
Identity {runIdentity = Just 2}
Prelude> runMaybeT $ (+1) <$> MaybeT (Identity Nothing)
Identity {runIdentity = Nothing}
```

Given that, we can get Identity from IdentityT and so on in the following manner:

```
type Identity 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.

---

[6]`https://wiki.haskell.org/Amb`

Yeah, but why? You don't ordinarily need to do this if you're working with a transformer that has an 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 out of that 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 intuitions with respect to the relationship it has with the structure of your values. Let us note something in the definitions of the following types:

```haskell
-- 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:

```haskell
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:

```haskell
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 just feed the unit value to the function:

```
Prelude> readerUnwrap ()
Right (Just 1)
```

Why is this the result? Consider that we used `return` for a Monad comprised of Reader, Either, and Maybe:

```haskell
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 "one-big-Monad" of Reader/Either/Maybe 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

Turn `readerUnwrap` from the previous example back into `embedded` through the use of the data constructors for each transformer.

```
-- Modify it 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 awhile with `Functor`, which lifts a function into a context (or, alternatively, lifts the function over the 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 of 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 in order to manage those stacks, first, we need to lift more.

The typeclass that lifts

MonadTrans is a typeclass with one core method: `lift`. Speaking generally, it is about lifting actions in some Monad over a transformer type which 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:

```haskell
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 you can see that 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 inide that 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 just how much is packed up in there.

Now, back to our example. This is the "hello, world" example using Scotty, but the following will cause a type error:

```haskell
-- 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 with this 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 expected type
  'Web.Scotty.Internal.Types.ActionT
   Data.Text.Internal.Lazy.Text IO a0'
  with 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, ....] }'
```

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:

```
(lift :: IO a -> ActionM a) (putStrLn "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 for ourselves what this `lift` did 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 definfition of ActionT:

```haskell
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:

```haskell
{-# LANGUAGE OverloadedStrings #-}

module Scotty where

import Web.Scotty
import Web.Scotty.Internal.Types (ActionT(..))
import Control.Monad.Trans.Class
import Data.Monoid (mconcat)

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:

```haskell
instance MonadTrans (ExceptT e) where
    lift = ExceptT . liftM Right
```

To use that in our code, add the following import:

```haskell
import Control.Monad.Trans.Except
```

And our app changes into the following:

```
main = scotty 3000 $ do
  get "/:word" $ do
    beam <- param "word"
    (ActionT
     . (ExceptT . fmap Right)
     . lift
     . lift) (putStrLn "hello")
    html $ mconcat ["<h1>Scotty, ", beam, " me up!</h1>"]
```

Then for ReaderT, we take a gander at Control.Monad.Trans.Reader 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 (?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:

```haskell
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've done:

```haskell
     . (\m -> ReaderT (const m))
```

Or:

```haskell
    (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 mentioned, we see the following Monad-Trans instance:

```haskell
instance MonadTrans (StateT s) where
    lift m = StateT $ \ s -> do
        a <- m
        return (a, s)
```

First, let's get our import in place:

```haskell
import Control.Monad.Trans.State.Lazy hiding (get)
```

We needed 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:

```haskell
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 needed an outer lambda before the StateT in order to get the monadic action we were 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 Maybe structure, we choose to not be obfuscatory and 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

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.

   ```
   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:

```haskell
addSubWidget :: (YesodSubRoute sub master) =>
                sub
             -> GWidget sub master a
             -> GWidget sub' master a
addSubWidget sub w =
  do master <- liftHandler getYesod
     let sr = fromSubRoute sub master
     i <- GWidget $ 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 $ unGWidget w
     let ((((((((a,
                 body),
               title),
              scripts),
             stylesheets),
            style),
           jscript),
          h),
         i') = w'
     GWidget $ 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. Especially, 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 actually 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:

```
Prelude> import Web.Scotty
-- again, to make the type read more nicely
-- we import some other modules.
Prelude> import Data.Text.Lazy
Prelude> :info ActionM
type ActionM = Web.Scotty.Internal.Types.ActionT Text IO
    -- Defined in 'Web.Scotty'
```

Hum. 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. Scotty's 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:

```
Prelude> import Web.Scotty.Internal.Types
-- more modules to clean up the 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 had 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 just fine:

```
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 the `main` function 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 used a browser and see "hello" printed more than once, it's highly 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

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 just some example of MaybeT in use; we will not comment upon them and instead let you research them further yourself if you want. Origins of the code are noted in the samples.

```haskell
-- 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)
```

```haskell
-- 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 ex-
tending additional structure to avoid boilerplate. Here's an example using
plain old Maybe and Scotty:

```haskell
{-# 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 returned `ActionM (Maybe ...)` and we wanted to short-circuit
the moment any of them failed. So, we do something similar but with
MaybeT and building up more data in one go:

```haskell
{-# 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"
    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 had to use `liftIO` once, even in the presence of additional structure, whereas with `lift` we had 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 what 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 really 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 was produced so that you avoid mysterious Nothing values floating around and short-circuiting your code. They're not something you really 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:

```haskell
{-# 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 had to manually fold the Either if we wanted 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 had many things we were pulling out of the parameters.

Let's do that but with ExceptT from **transformers**. Remember, ExceptT is just another name for EitherT:

```haskell
{-# 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:

```
http://localhost:3000/?1=1
```

It'll ask for the parameter `2` because that was the next param you asked for after `1`.

If you pass it a request like:

```
http://localhost:3000/?1=1&2=2&3=3&4=4
```

You should see the response in your browser or terminal of:

```
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[7] by Sean Chalmers[8].

Some context for the `EitherT` application you'll see:

---

[7]`https://github.com/mankyKitty/Meteor/`
[8]`http://mankykitty.github.io/`

```haskell
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
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:

```haskell
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 -- no missiles at start
                     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*!

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 not to get stuck in modern Haskell libraries or frameworks, but it's not a design dictate you must follow.

In a later chapter we'll be showing you something that you might find more appealing or nicer than concrete monad transformers, but you still shouldn't consider that an invitation to premature elaboration.

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 actually ReaderT of Identity and that `runReader` is a convenience function throwing away the meaningless structure for you. Play with `runReaderT` if it tickles your nondescript furry red puppet.

2. Once you have an `rDec` that works, make it and any inner lambdas pointfree if that's not already the case.

3. `rShow` is `show`, but in Reader.

```
rShow :: Show a => ReaderT a Identity String
rShow = undefined

Prelude> runReader rShow 1
"1"
Prelude> fmap (runReader rShow) [1..10]
["1","2","3","4","5","6","7","8","9","10"]
```

4. Once you have an `rShow` that works, make it pointfree.

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 typecheck as written; fix it so that it does. Feel free to add imports if it provides something useful. Functions will be 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.

```haskell
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 which counts hits to specific URIs. It also prefixes the keys with a prefix defined on app initialization, retrieved via the command-line arguments.

```haskell
{-# 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
```

```haskell
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, 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 passed "lol" to `main`. For a giggle, try the URI for one of the keys in your browser and mash refresh a bunch.

Morra

1. Write the game Morra[9] using StateT and IO. The state being accumulated is the score of the player and the computer AI the player is playing against. 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 hotseat 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
   ```

---

[9]`https://en.wikipedia.org/wiki/Morra_(game)`

```
C: 1
- P wins
```

At this point, the computer should register the pattern (1, 2, 2) player picked 2 after 1 and 2. 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 still bad at being random; they often have sub-patterns in their moves.

26.15 Answers

rShow

```haskell
rShow :: Show a => ReaderT a Identity String
rShow = ReaderT $ return . show
```

HitCounter

Just a hint: use `atomicModifyIORef'` to get the counter state, update it, and get the new state outside of the update in one go.

26.16 Follow-up resources

1. Parallel and Concurrent Programming in Haskell; Simon Marlow; http://chimera.labs.oreilly.com/books/1230000000929