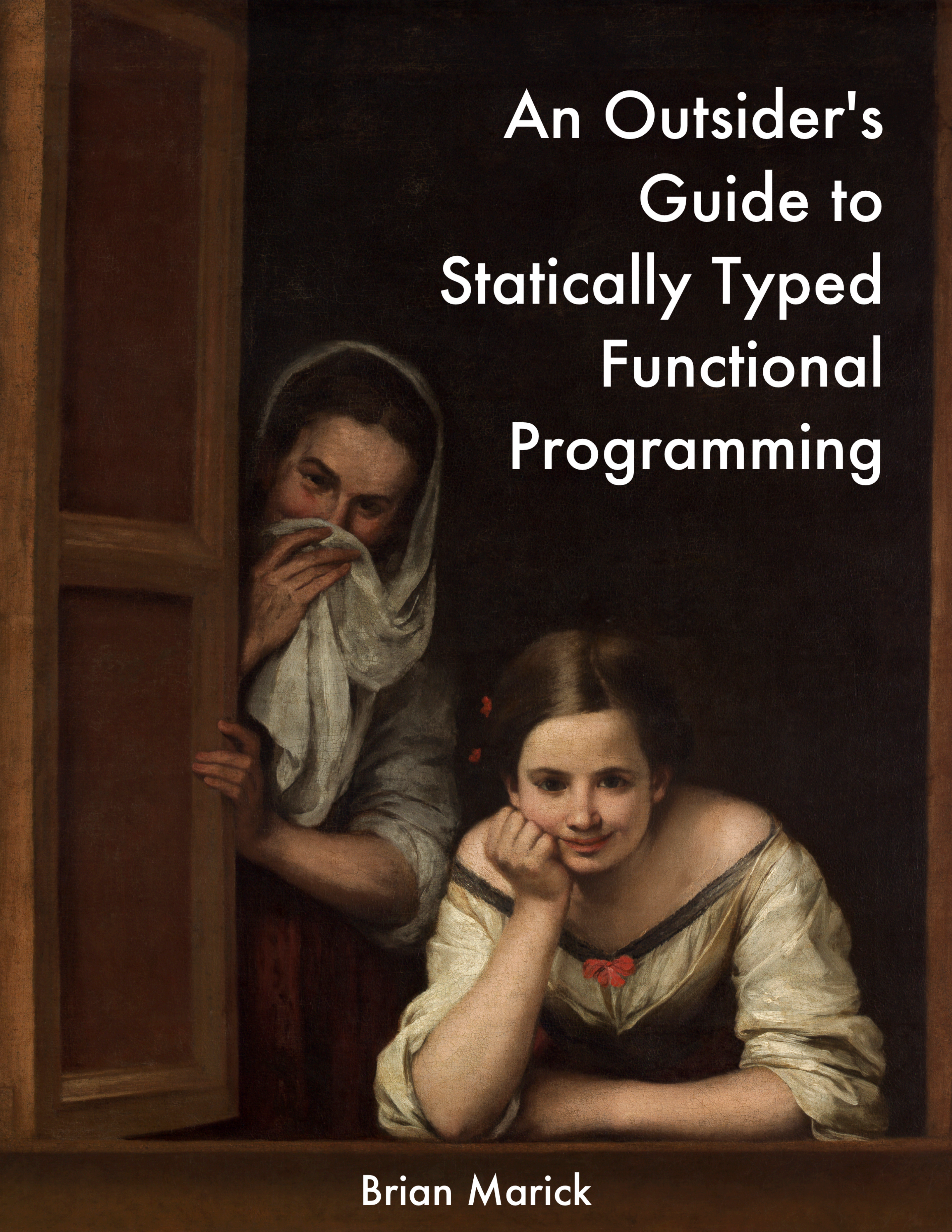


An Outsider's Guide to Statically Typed Functional Programming

Brian Marick

An Outsider's Guide to Statically Typed Functional Programming

Brian Marick

This book is for sale at http://leanpub.com/outsidefp

This version was published on 2018-04-05

Contents

Introduction

Our situation is analogous to that of someone who has learned the rules for how the pieces move in chess but knows nothing of typical openings, tactics, or strategy. Like the novice chess player, we don't yet know the common patterns of usage in the domain. We lack the knowledge of which moves are worth making…

– Harold Abelson and Gerald Jay Sussman, *Structure and Interpretation of Computer Programs*

Merrick… Merrick wanted to put here [stabs forefinger at newspaper] "gratis." Now, is the idea to inform your reader or make him feel like a [bad word]ing dunce, huh? I had him put "free."

– Al Swearengen, in "Deadwood," Season 1, Episode 6: "Plague"

Why bother with static functional languages?

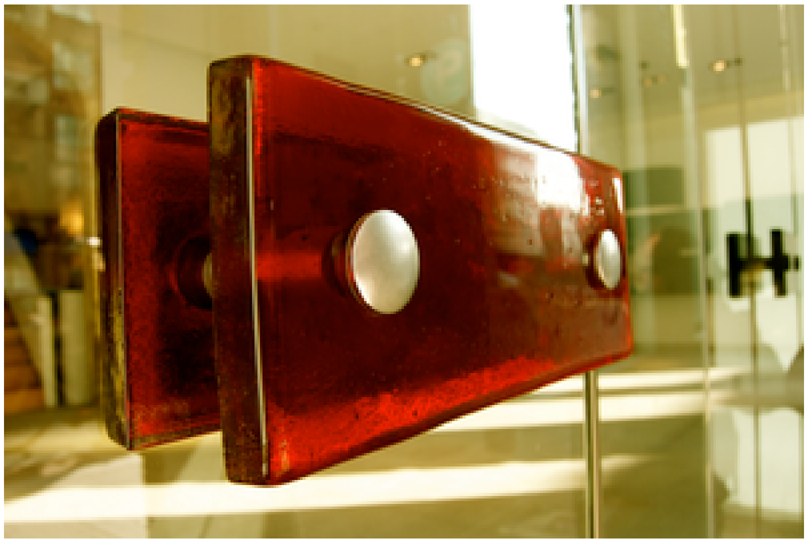
You've probably read various claims that programs written using statically typed functional programming (hereafter "static FP") cannot have certain kinds of runtime errors. Sometimes those claims have an unpleasantly moralistic slant: you are a *bad person* for willingly writing in a language that allows null pointer exceptions.

I'm not going to make that argument. I prefer to motivate via excitement than via shame, and I think there's a different argument that might motivate you more. It's about *affordance.*

In psychology, affordance means the kinds of actions a physical object invites.[1] For example, a big flat door handle like the one in the following picture "affords" pushing to open it:[2]

[1]The psychologist James Gibson coined the term in 1977, and described it more fully in his *The Ecological Approach to Visual Perception.* It's an interesting book.

[2]This image and the next are from Marc Falardeau and are covered by the Creative Commons Attribution 2.0 Generic license.



Push me

… and a vertical handle that you can wrap your hand around says, “Pull me!”:

Pull me

There’s a problem with each of those pictures, though: the handles are the same on *both sides of the door*, so the affordance is backwards for one side. I bet we’ve all had the experience of pushing at a door only to discover we should have been pulling. It’s annoying. But we all probably go through the door anyway. What other choice do we have?

However, when it comes to user interfaces, affordance can greatly change behavior. If some action isn’t “afforded,” it often doesn’t get done. Since a programming language is the programmer’s interface to the computer, what your language affords matters a lot.

Suppose, for example, you're writing some lines of code that work with the first element of a list. You *know* that the list can never be empty. At least, you're pretty sure that there's no path through the whole program that allows an empty list to get to the code you're writing. Still, maybe you're nervous. You could check for an empty list and then… do what? Most likely, you can't do much better than what happens if code tries to work with the first element of an empty list: some exception (probably a null pointer exception) gets thrown. Is it really worth bothering to make a check that can only shout – perhaps slightly more clearly – that something supposedly impossible happened?

Alternately, you could be more ambitious and construct your own flavor of list that would always be non-empty. So if any code ever tried to take off the last element, your list data structure would… would… do what? Probably throw an exception. Again, you're not really better off.

It's just not clear that a typical language gives you compelling reasons to deviate from the simplest tactic: ignore what you think won't happen.

In constrast, a static FP language lets you create a list that literally *cannot* be empty, ever. What used to be a matter of what you believe becomes a matter of what the compiler guarantees. So you don't have to worry about what to do if an empty list arrives. It won't happen.

Moreover, static FP languages don't only force you to check the "nothing here, boss" case (which is what `null` usually means), they also afford a way of doing it that's less obnoxiously branchy than it would be in many languages. (If you know Clojure, it's reminiscent of the `some->` macro.)

So…

The reason to bother learning static FP languages is to discover what code they afford that would be pointless or too hard in other languages. When it comes to how apps evolve over time, such affordances matter.

All languages afford some things and discourage others, so you should also expect this book to be frank about what the static FP languages "dis-afford." After reading it, you should be able to understand – at a practical, somewhat experienced level – the pros and cons of both your current languages and the static FP languages. Then you can decide which is a better fit for your needs.

Why is this book called an "outsider's" guide?

It's an outsider's guide because I'm outsider, both temperamentally and in fact.

A lot of what I do with my life is find fields that others have studied deeply, plunge into them, and study them… less deeply. (I'm OK with "shallowly," too.) My goal is not to contribute to the field as an insider, but rather to harvest ideas, tools, and techniques that will help me create computer programs – and help me advise others who are creating computer programs. Static FP is the latest of those fields.

I'm also an outsider to FP because I have some peculiar opinions:

1. Abstraction is a wonderful tool, but it's nothing but a tool.

2. Interesting designs are those that handle messy domains. Messy domains are those where abstractions are useful but insufficient. I'm interested in the abstractions, but I'm *more* interested in how the design accommodates exceptions and mis-fits.

   Although object-oriented languages are widely derided in FP circles, much literature on object-oriented design has been (sometimes implicitly) about how to make models when you can't force the world outside the program to act in a way that fits the abstractions we've learned to love. I believe the FP world can and should learn from that.
3. The end result of a design is less interesting than the process by which it was created. The idioms and habits that make up the day-to-day life of a programmer are important knowledge that should be more widely taught.
4. Mathematicians are, by and large, *horrible* at explaining things, and they're particularly bad at naming. Some of what static FP has taken from mathematics is foundational and necessary. But FP has also aquired cultural shibboleths[3] that unnecessarily repel people.

I believe that writing about static FP from a deliberate outsider's perspective will produce a book useful to people who've found something missing or off-putting in existing FP books.

I also have a sneaking hope that this book will persuade some insiders to look more closely at points 2 and 3.

What the book covers

The book teachs the ideas and idioms of statically-typed functional programming via three progressively more powerful languages. (Importantly, all three languages have very similar syntax.)

It begins teaching them using the Elm programming language. Elm is one of the breed of languages that compiles down to JavaScript. It's also – as far as I know – the friendliest of the static FP languages. Its error messages are superb. Its documentation is good and emphasizes plain language over jargon. Its focus is on the beginner, and the features it provides are well-chosen for that audience. Its command-line interface (the "repl"[4]) tries hard not to surprise you with puzzles you have to figure out. It's robust enough that I felt (mostly) confident using it in two real-world projects – and my confidence was not misplaced.

However. (There's always a "however".)

Elm's laser focus on beginner understandability and usability means that it's missing features or concepts that other static FP languages provide. In particular, it provides extremely limited *polymorphism*: that feature that lets programmers give a single name to two different functions or methods, confident that the compiler or runtime system will figure out which implementation is

[3]Per Wikipedia, a shibboleth "is a word or custom whose variations in pronunciation or style are used to differentiate members of ingroups from those of outgroups, with each receiving value judgments of superior or inferior."

[4]"Repl" started out as an acronym for "read-eval-print-loop," but it's become a word in its own right. These days, many languages have repls. They let you type snippets of program and immediately see what they do.

right for each use of the name. Polymorphism is important because it lets programmers say "these two things are in some sense the same, and in some sense different. Here's how…". In pure object-oriented languages, you get polymorphism from inheritance. Other languages have what I think of as variations on the same theme: traits, multimethods, or protocols.[5] In static FP, there's an interestingly different approach.

I'll show what's missing from Elm by teaching Purescript. Purescript also compiles to JavaScript. It implements important features from the mainstream of static FP. It isn't (yet) as good as Elm at documentation, error messages, or the repl, but the people behind it seem to understand their importance.[6] That makes it the gentlest introduction to the more abstract and mathematical parts of static FP.

I may – I'm not sure yet – also cover Idris. Idris takes the whole "static typing" idea up another level. In Elm and Purescript, types exist entirely separately from your code, which computes only on the concrete values your types allow. Idris lets your code compute on types *themselves*. That is, Idris lets you have not only a function that checks equality, but also a function that returns the abstract concept "equality check."

Although Idris doesn't have the immediate applicability of Elm and Purescript (with their focus on living within the JavaScript ecosystem), I'm excited by its potential.

How the book covers it

There are a number of topics that any book about static FP has to cover. The topics have dependencies between them: some fact about topic `F` depends on some fact about topic `B`. The `B` fact has to be explained before the `F` fact. The easy way to do that is to have a chapter about `B` that includes all its facts, and then later a chapter about `F` that contains all *its* facts. As an author, you only have to keep track of whole topics, not the much larger number of individual facts. Also, that organization makes it easier to use the book as a reference. (If the reader half-remembers a fact about `B`, she knows what chapter to look in.)

That would be an ideal book to be read by a machine with no goals other than to (1) absorb all the facts in the book, (2) not be confused by an undefined term, and (3) perhaps rigorously check to see that later facts are correctly making use of earlier ones.

You are not such a machine. Your goal is to write programs, not absorb facts. Writing programs requires taking topics like `A`, `F`, and `M` and combining them in a piece of code – not because of their logical relationships but because of what you want your program to accomplish.

Here's an example. Later in the book, you'll learn about "sum" and "product" types. For now, you only need to know that both can be nested, either within themselves or within each other. When I was

[5] Traits in Scala. Traits in Rust. Multimethods originally come from Common Lisp and are also available in Clojure. Idiomatic Clojure uses protocols.

[6] For way too long, the dominant FP cultures – both static and dynamic – gave (me, at least) the impression that you had to *earn* the right to use their languages. And you earned it by suffering. ("The beatings will continue until morale improves.") This is changing rapidly.

working with form data in an Elm app, I kept facing questions like how to model a form's original data, its changed data, and any "flash" messages to the user. I tried sum types inside product types, product types inside sum types, and all kinds of variations. For a while, it seemed like each new feature would make me change my form model, causing me much wailing and gnashing of teeth.

Understanding the two topics (the two kinds of types) separately wasn't enough. That's why there's a chapter specifically on combining them.

Adding a new chapter was an easy solution for that particular case, but many others don't have easy solutions. Consider this progression:

1. There are sum types. Here are facts about the code that defines and manipulates them.
2. `Maybe` is an example of a sum type.
3. `Maybe` is the way you avoid null-pointer exceptions.
4. There are "functors." Here are facts about code that uses them.
5. `Maybe` is an example of a functor.
6. Because of that, there's a way to use `Maybe` that isn't infuriatingly tedious.

As a logical structure of facts, that's fine. But it'd be pretty mean of me to structure the book that way. There's necessarily a big gap between topics 3 and 4. You'd get to feel unhappy about `Maybe`'s tediousness for many chapters before you got to see how to avoid it.

So I'm using a different structure:

1. Chapter 5 is about `Maybe`. It's the way you avoid null-pointer exceptions.
2. It includes a way to use `Maybe` that isn't infuriatingly tedious.
3. It includes other idioms for using `Maybe`.
4. Chapter 7 is about sum types. It gives you a deeper understanding of what you learned to do with `Maybe`.
5. A much later chapter is about functors so that you can understand `Maybe` even better.

That is, the book is organized around feelings more than facts. As much as I can, I want you to leave each chapter feeling you've more power at your disposal, that your ability to do some kinds of things has been increased.[7]

Let me put it differently. Much of the intellectual fascination of static FP comes from how it lets you create powerful abstractions – building blocks – using two basic moves: combining functions and combining types. This book, though, will cover those foundational topics superficially to make room for common ways of combining the *resulting* building blocks. Where possible and reasonable, it'll then combine those combinations, moving from smaller-scale program design to what could be called architecture, including the architecture of messy systems.

[7]Some philosophers make a distinction between "knowing that [something is true]" and "knowing how [to accomplish something in the world]." This book comes down heavily on the side of knowing how.

Prerequisites

You should be a programmer.

- I'll sometimes refer to object-oriented programming, but nothing essential depends on you knowing it. When I show snippets from an object-oriented language, they'll be short and simple. I'll use Ruby or Java for them.
- Don't worry if you only know a dynamically-typed object-oriented language like Ruby. The static types in, say, Java are different enough that there's no advantage to knowing about them.
- If you know a dynamically-typed functional language like Clojure or Elixir, you'll already be familiar with some minor topics like immutability. You're likely to be more comfortable with recursion or list processing than an OO programmer. But the heart of the static FP languages is the type system, and it makes the experience of programming much different than in the dynamic FP languages.

Your life will be much easier if you're familiar with a programmer's editor – one that understands a language well enough to help you with indentation and to colorize text according to syntax. The book's languages have plugins for:

- the Big Two traditional editors: Emacs and vim.
- the new kids on the block: Atom and Sublime Text.
- Visual Studio and sometimes IntelliJ IDEA.

If you don't have an editor already, I understand that both Sublime Text and Atom are easy to learn.

The exercises

Exercises are scattered throughout the chapters. I urge you to try them; if you're like most people, you'll learn a lot you otherwise wouldn't.

Early readers: I think there are still too few exercises, especially in the early chapters. Let me know if you find yourself wishing that an earlier exercise had helped you clarify an idea. Although most of the exercises are coding exercises, I'm also looking for excuses to write "essay questions." ("This function produces this result. Can you explain why?")

All of the book's source code is stored on GitHub. Exercise solutions are stored in the repository's wiki. For longer exercises, the wiki will contain only the key code. If so, there will be a link to the complete, executable source.

Feel free to add your own comments to wiki pages. I may edit or delete them if they're superseded by later versions of the book.

The back of the book

I almost never read the table of contents when I read a book, so I often don't notice useful appendices until I turn the last page. This book has a glossary and (incomplete) language references. Terms in the glossary are *italicized* when they're first defined.

About the cover

The cover image is *Two Women at a Window*, by Spanish painter Bartolomé Esteban Murillo (1617–1682). While primarily a religious painter, he's also known for his paintings of the street life of his cities (primarily Seville and Madrid).

I chose that painting for the cover because I imagine an unseen suitor wooing those two women, feeling unsettled by being giggled at and – worse! – being viewed with politely-amused boredom. As a lifelong troublemaker and iconoclast, I get that kind of reaction a lot. And so I imagine those two women as FP insiders, seeing this book as unrigorous clownish capering, what with the idioms and messy systems and all.

And yet…

And yet…

I *think* the woman on the lower right is beginning – just beginning – to think kindly of her suitor. So she gives me hope this book will find favor with both outsiders and insiders.

Your problems and questions

For the moment, either send problems to me or post issues in the Github repo.

Please include the version of the book. This is **version seven**.

Change log

version seven

- Finished the chapter introducing PureScript.
- Added that chapter to the sample, together with relevant parts of three prerequisite chapters (about typeclasses and JSON).
- Finished the "Finer-grained access to the outside world" chapter.

version six

- Finished Lenses and Laws with a chapter on using lenses for error handling and one that uses JSON decoders to introduce the idea of “functors”.
- In the source:
- The `Monoid` directory has been renamed `TypeClass`.
- Some renaming in the lens code. The `Try3` module has been renamed `Final`. `Dict.lens` is `Dict.upsertLens`. `Array.lens` is `Array.humbleLens`. `Lens.Try3.Compose.Operators` is now `Lens.Final.Operators`.
- There’s a new type of lens in `Lens.Final.Lens`: the `Path` lens. It’s explained in the chapter on error handling.
- Reworked the lens tests to make them more comprehensible. Instead of *n* test modules, each of which tests all the lenses, there’s a single test module for each lens.

version five

- Added chapters 23-25, the first three chapters of Lenses and Laws.

version four

- Added The Saga of the Animation App, chapters 15 through 22.

version three

- Some exercises were added to the heterogeneous data section of the chapter on sum type idioms.
- A new chapter on records, plus a new set of chapters on a restricted problem that’s messy because of error handling.

Thanks for the help

These people provided helpful comments on drafts or answered my questions:

Adam Solove
Agile Hulk
Alex Berg
Bill Wake
Chris Ford
Colin Yates
Devin Walters
Fabio Akita
Gabe Johnson

Gary Burgess
Hardy Jones
Harry Garrood
James Byatt
Jason Stiebs
Jeremy W. Sherman
Joe Kachmar
Johannes Link
Luis F. Gutiérrez
Mark Windholtz
Mattox Beckman
Nicholas Scheel
Rahul Goma Phulore
Suresh Iyer
Xavier Lepaul

I apologize if I overlooked someone.

Special thanks to my informal “advisory board”:

Bill Tucker
Bill Wake
J.B. Rainsberger
Keith Braithwaite
Nat Pryce
Sallyann Freudenberg
Steve Freeman

Assume they vigorously objected to all of my bad decisions about the book.

Also thanks to Brad, Seth, Jesse, Rich, Tim, Ken, Tricia, and Sullivan of the Green Street Potbelly’s, who kept me well supplied with Diet Coke.

Finally, I’d like to thank dead physicist Erwin Schrödinger. The accent in “Bartolomé Esteban Murillo” showed up as a garbage character until I placed the text “Schrödinger’s” (apostrophe required) in this chapter. There’s a good quantum mechanics joke in there somewhere, and it makes the problem too entertaining to fix.

I Elm

Installation

The main Elm site is http://elm-lang.org. Installation instructions are here: https://guide.elm-lang.org/install.html.

After installing, do pay attention to the links to editors that can be made to understand Elm. We'll do some work in the repl, but once you write larger functions, you'll want to edit files. Like all the languages we'll cover, Elm has significant indentation. That is, you'll get no complaints about this:

```
something bool =
  case bool of
    True -> 5
    False -> 6
```

... but this:

```
something bool =
  case bool of
    True -> 5
   False -> 6
```

... will produce an error message. It might be clear to you. It might not. It's better if your editor prevents you from ever making the error.

You can stop reading the installation instructions when you get to the documentation on the various Elm commands.

The book's repository

Visit the book's Github repository and clone it.

Go to the `elm` subdirectory and type `elm-package install`. That will install a few non-default packages. For Elm to find them, you'll have to do all your work within the `elm` subdirectory.

Elm documentation

Elm comes with pre-built functions, lumped together as "core." It has API-style documentation. Elm has a package manager. Each package will have API documentation and perhaps other documentation as well.

1. Functions

Given the name "Functional Programming," it seems only right to start with functions.

Applying functions

Start the Elm repl (with `elm-repl`) and type the following. Don't worry if the repl takes some time to print the answer. It's phoning home to see if it needs to download anything.

```
% elm-repl
> List.take 2 [1, 2, 3]
[1,2] : List number
```

That's how you call (*apply*, in the jargon) a function in Elm. Here are some things to notice:

1. Functions live in *modules* with names like `List`. Modules are just a way to collect related names. You can think of them as like Java classes with only static members, or as like Ruby modules. They're not like regular classes because they don't contain or describe any changeable instance variables.

   For clarity, you'll often use the module name in a function call, but you don't have to if you *expose* the function:

   ```
   > import List exposing (take)
   > take 2 [1, 2, 3]
   [1,2] : List number
   ```

2. A function's *arguments* are space-separated, rather than the comma-separated form you're probably used to. Parentheses need only be used when there are nested function calls. Here's an example. If you type the following:

   ```
   > List.take 2 List.append [1] [4, 5, 6]
   ```

   … you'll see an error message that you've given `List.take` four arguments when it expected two. The four arguments are the number 2, the function `List.append`, and the two lists.

   Here's the parenthesized version:

```
> List.take 2 (List.append [1] [4, 5, 6])
[1,4] : List number
```

   (There are two other ways to write nested calls. They'll be covered below.)

3. The repl not only prints the result (`[1, 4]`) but its *type* (a list of numbers). We'll see about types in the next chapter, so ignore them in this one.

The parenthesized example uses `List.append` to concatenate two lists. You can also use an *operator* to do the same thing:

```
> [1, 2, 3] ++ [4, 5]
[1,2,3,4,5] : List number
```

`++` is nothing but "syntactic sugar" that lets you put a function between two arguments (rather than in front of them). Other than that, `++` is a function just like `List.append`. In fact, you can tell Elm to treat an operator as a normal function by wrapping it in parentheses:

```
> (++) [1, 2, 3] [4, 5]
[1,2,3,4,5] : List number
```

That turns out to be handy, as you'll see throughout the book.

Prefix notation doesn't only work with weird operators like `++`, but also with more familiar ones like addition:

```
> 1 + 2
3 : number
> (+) 1 2
3 : number
```

Exercises

1. Predict the value of this expression:

   ```
   1 + 2 * 3
   ```

   Now use the function forms of `+` and `*` – that is, `(+)` and `(*)` – to write an expression that produces the same result.
2. What are the results of the following expressions?

```
    List.take 0 [1, 2, 3]
    List.take 4 [1, 2, 3]
    List.take -1 [1, 2, 3]
    List.take 1.3 [1, 2, 3]
```

Discussion and solutions

Defining functions

Here's the least convenient way to create a function:

```
> \ x y -> x + y
<function> : number -> number -> number
```

That's an *anonymous function*. It starts with a backslash that's supposed to remind you of λ, the lowercase Greek letter lambda. It has two *parameters*. The body of the function (after the `->`) adds them together.

Notice that there's nothing resembling a `return` statement. There's no need for one, as the entire body of the function is a single expression. Whatever it calculates is what's returned. (You'll see later in this chapter that there are ways to create something that looks less like a single expression and more like a series of statements.)

That function can be used in the repl by applying it to some arguments:[1]

```
> (\ x y -> x + y) 1 2
3 : number
```

The parentheses are needed for Elm to be able to tell where the anonymous function ends and its arguments begin.

It would be nice to give functions names. The inconvenient way to do that is this:

```
> sum = \x y -> x + y
<function> : number -> number -> number
```

In the jargon, that's called *binding* a function to a name. The phrase "naming a function" would be more familiar but also worse, as it implies that functions can have names. They don't. Instead, they're "pointed to" by names. You can see the distinction with the following two bindings, this time of numbers rather than functions:

[1]I'll use "arguments" to refer to the actual values a function computes with, and "parameters" for the corresponding names in a function's definition. That's not because I think there'll ever be a case in this book where calling both "arguments" would be a problem. It's more of a nostalgic shout-out to a much earlier version of me who thought such precision *Really Important*.

```
> aBitLessThanFive = 4
4 : number
> myFavoriteNumber = aBitLessThanFive
4 : number
```

Neither of those steps gives a new name to `4`; they just give you a way to refer to the same underlying value. The same is true when binding a function.

For convenience, Elm lets you move the arguments to the other side of the `=` sign, producing a definition that's less cluttered and more clearly reminds you of how the function's used:

```
> sum x y = x + y
<function> : number -> number -> number
```

Let's try a different, slightly more interesting function:

```
> combine combinerFunction x y = combinerFunction x y
<function> : (a -> b -> c) -> a -> b -> c
```

I want to emphasize that, in functional languages, functions are values, just like numbers are. So you can pass them as arguments to other functions. Here is how you could use `combine`:

```
> combine (+) 2 3   -- combine 2 with 3, using +
5 : number
> combine (-) 2 3
-1 : number
```

To understand how those lines work, replace uses of parameters in the body of the function with the actual arguments. That is, given the earlier definition:

```
combine combinerFunction x y = combinerFunction x y
```

… and the use `combine (+) 2 3`, `combinerFunction` is replaced with `(+)`, `x` with `2`, and `y` with `3`. The result is this:

```
> (+) 2 3
5 : number
```

Um, that's not actually useful at all, since you could more easily type `(+) 2 3` than the form using `combine`. I wanted to start with a simple case.

Probably the most common *useful* function that takes a function as an argument is named `map`. Here's an example of `List.map`:

```
> List.map String.length [ "nineteen characters", "four" ]
[19,4] : List Int
```

What happened here is that `List.map` applied `String.length` to each of the strings within the two-element list of strings and wrapped all the results up into a new list.

But lists aren't the only things that contain other things, and the idea of `map` applies more broadly. Just as lists contain elements, strings contain characters. And just as strings have functions (like `length`) that work on them, characters have functions that work on them. So it makes sense that there's a `String.map`.

Before using it, there's a tiny bit of preparation. Elm automatically makes the `String` module available, but you have to explicitly ask for `Char`. Like this:

```
import Char
```

Now we can use `Char.toUpper`, a function that uppercases a character:

```
> String.map Char.toUpper "aBAb"
"ABAB" : String
```

The jargon for this is "mapping <a function> over <something that contains other things>". Thus, in a fully functional household, you might hear "Let's spend our vacation mapping `Char.toUpper` over strings!"[2]

Both versions of `map` are doing something similar: running linearly down a sequence of things, transforming each according to a function, and pasting the results back together. This looks like an opportunity for abstraction! It is, but it's too early to talk about that. There's more to learn about just plain functions.

Exercises

1. Define a function, `add5`, that adds `5` to its single argument. Use `sum` (from the beginning of this section).

   ```
   > add5 33
   38 : number
   ```

2. Use `List.map` to add `5` to a list of integers.
3. Make a function that takes a function and maps that function over the list `[1, 2, 3]`. You can use `add5` and the built-in function `negate` to test it, as shown here:

[2]Now you see why I married a veterinarian.

```
    > doTo123 add5
    [6,7,8] : List number
    > doTo123 negate
    [-1,-2,-3] : List number
```

Discussion and solutions

Partially-applied functions

There's a lot of power in passing functions around, as you'll see throughout the book. But there's even more power because functions in Elm can be *partially applied*. Here's what that means.

Consider our earlier `combine`:

```
> combine combinerFunction x y = combinerFunction x y
```

We saw what happens when it is applied to three arguments:

```
> combine (+) 2 3
5 : number
```

What if it's only applied to two?

```
> combine (+) 2
<function> : number -> number
```

The result isn't a number; instead, it's a function. Without diving too much into how to read the type `number -> number`, let's just say that the resulting function takes a number and produces a number. What does that function do? Well, let's bind the partially applied function to a name:

```
> add2 = combine (+) 2
<function> : number -> number
> add2 3
5 : number
```

We can also leave off `combine`'s last two arguments, like so:

```
> plus = combine (+)
> minus = combine (-)
> plus 1 2
3 : number
> minus 1 2
-1 : number
```

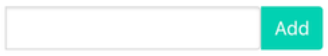
Note: I'm sticking with `combine` to emphasize that, in functional programming, functions are values like everything else. Your code will spend a *lot* of time creating new ones and passing them around. But don't forget you can partially apply `(+)` itself:

```
> add2 = (+) 2
<function> : number -> number
> add2 3
5 : number
```

You can think of this partial application as creating a new function by "pre-supplying" the first of the two arguments that `(+)` works with.

There'll be more about how partial application works in the next chapter.

Partially applied functions are enormously common in Elm code. Consider this HTML control from one of my apps:



I create it using a functional version of the Builder pattern. In the following code, pay attention to line 5. We'll see more about `|>` soon. For now, just think of a structure being created on line 1, then flowing through a series of steps, accumulating values as it goes.

```
1      Css.freshValue form.tentativeTag
2        |> TextField.editingEvents ...
3        |> TextField.eventsObeyForm form
4        |> TextField.kind TextField.errorIndicatingTextField
5        |> TextField.buttonKind (Button.primaryButton "Add")   -- <<<<<<
6        |> TextField.build
```

The definition of the `Button.primaryButton` function has two parameters: the text to show in the button and a list of JavaScript event handlers. Only one (the first) argument is given on line 5 because the event handlers to use are implied by the earlier steps. Line 6's final step (`TextField.build`) applies the partial function from line 5 to the implied event handlers, finally producing an actual button that works with a text field.

The neat thing about `Button.primaryButton` is that it's a function that *automatically* affords two uses:

1. A partially-applied version fits nicely into builders that construct aggregate visual controls.
2. A fully-applied version is an easy way to create a button that stands alone:

   ```
   Button.primaryButton "Click to add more information" buttonEvents
   ```

Exercises

1. What does this function do?

   ```
   > reversi x f = f x
   ```

   Hint: try `negate` or `add5` as one of the arguments.
2. Given this:

   ```
   > mystery = reversi 3
   ```

   … what does `mystery` do?
3. What is the result of this:

   ```
   > List.map mystery [negate, add5]
   ```

Discussion and solutions

Idiom: Self goes last

Because it's so common to pass partially-applied functions to other functions that "fill in the blanks" of omitted arguments, you should define your functions with that in mind. That is, for any function `f first second`, the first argument should be the one that you're most likely to want to pre-supply, and the second should be the one you're most likely to want a receiving function to supply.

`List.take` fits that. If you're going to create a partially applied function, it's much more likely you'll want to say "take the first five elements of a variety of lists":

```
someReceivingFunction (List.take 5) maybeMoreArgs...
```

… than "take some number of elements of this specific list [1, 2, 3]":

```
someReceivingFunction (List.bizarroTake [1, 2, 3]) maybeMoreArgs...
```

… and that’s why `List.take` has the argument order it does.

Now, while predicting the future is easy, getting the prediction *right* is harder. Guidelines help. One is that collection arguments (lists, arrays, dictionaries, trees, etc.) should usually go last. You’ll find that rule followed in Elm’s built-in modules.

Another guideline I find helpful is to think in terms of object design: which argument would you make the `this` or `self` of a method call? That is, would you expect the Ruby `Integer` class to have a `take` method that took a list? Or would you expect `List` to have a `take` method that took an integer? The `take` method is much more “about” lists than about integers, so the list should be the last argument.

Flip: a function that only cares about functions

Elm comes with a function named `flip` that’s very similar to `combine`. Here’s the definition of `combine` again:

```
> combine combinerFunction x y = combinerFunction x y
```

Let me rename the parameters:

```
> combine f first second = f first second
```

… and then change `combine`’s name and make one tiny change to its definition:

```
> combine f first second   = f first  second   -- old
> flip    f first second   = f second first    -- new
```

Interesting! Using `flip`, we can apply a function to its two arguments in the opposite order. Compare these two ways of using `List.take`:

```
>      List.take        2       [101, 202, 303]
[101,202]
> flip List.take [101, 202, 303]       2
[101,202]
```

Well. That seems about as useless as `combine`, and it would be – were it not for partial application. Partial application is about pre-supplying arguments, but you have to do that beginning with the first argument. What if you want to pre-supply the second? `flip` to the rescue:

```
someReceivingFunction (flip List.take [1, 2, 3]) someOtherArg
```

`flip` makes it easier to pass partial functions in the uncommon case where “self” wasn’t the right final argument.

Note: Although we defined `flip` in this section, Elm predefines it for you.

Exercises

I found partial application and `flip` confusing at first. If you’re like me, maybe these exercises will help.

1. What does the following code produce?

   ```
   > flip (++) [1, 2] [3, 4]
   ```

2. In what follows, I’ve given a bad name to the function created via a partial application of `flip`. What would be a better name?

   ```
   > mystery1 = flip (++)
   ```

   (Recall that `++` is an operator that does the same thing as `List.append`.)
3. What’s the result of the following?

   ```
   > List.map (flip (++) [1, 2]) [[9], [99], [9,99]]
   ```

4. Here’s another bad name. What would be a better one?

   ```
   > mystery2 = flip (++) [1, 2]
   ```

5. `flip` works on functions that take two arguments. Here’s a simple function that operates on three arguments:

   ```
   > list3 a b c = [a, b, c]
   > list3 1 2 3
   [1,2,3] : List number
   ```

   How can you use `flip` to flip the second and third arguments? That is, construct some combination of `flip` and partial application and parentheses to produce this:

```
> ... list3 ... 1 ... 2 ... 3 ...
[1,3,2] : List number
```

This kind of fancy footwork is usually a bad idea in code other people will read. It would be better to make a new function. It's no accident that there's no `flipArgs2and3of3TotalArgs` in the Elm libraries.

Hint in a footnote.[3]

Discussion and solutions

Point-free style

Consider a function that negates every number in a list:

```
> negateAll [1, 2, 3]
[-1,-2,-3] : List number
```

Here are three ways to bind the name `negateAll` to the function that does the work:

```
1  > negateAll = \list -> List.map negate list
2  > negateAll list = List.map negate list
3  > negateAll = List.map negate
```

1. This binds the name to an anonymous function.
2. This is the more convenient syntax for the same thing.
3. This binds the name to a function created by partial application. The `list` argument can be omitted. This style rejoices in the name *point-free style*.[4]

It makes no difference to Elm which style you use. Point-free definitions can get elaborate and clever. When should you add named parameters for clarity?

My bias is toward named parameters. To my way of thinking, point-free style is appropriate when:

1. You're sure that named parameters will not help a later reader understand the function.

[3]You can use partial application to create a two argument function, then apply `flip` to it.

[4]Editorial: although I love functional programmers and designers of functional languages absolutely to death, they overdo it with analogies to mathematics. "Point-free" is apparently justifiable by reference to topology. While there are sometimes good reasons for using abstract terminology, this is emphatically not one of them. "A parameter-free function definition" would have been way better than "a point-free function definition."

2. You're willing to sign an affidavit testifying that you understand and agree that "a later reader" may be less experienced than you and certainly will not know everything you know at the moment you wrote the function.
3. You do not feel smug about the coolness or cleverness of your point-free definition.[5] (This is the most common reason why I rewrite my point-free definitions to use named parameters.)

Function composition

`flip` is so often partially applied that I mainly think of it as a function that converts an existing function into a new one. Two other useful functions convert *two* existing functions into a new one. They're invariably written in operator form: >> and <<.

To talk about these new operators, let me define three functions:

```
add2 = (+) 2
times3 = (*) 3
minus8 = flip (-) 8
```

Oops… I just violated my own heuristic about point-free definitions. I felt a little thrill of pride writing `minus8` that way. Which means I should write it this way:

```
> minus8 x = x - 8
<function> : number -> number
```

How would we create a function that combines (or *composes*) these three functions? We could do this:

```
silly x = minus8 (add2 (times3 x))
```

That works, but it's awkward for speakers of most Western languages. We're used to both words and time flowing from left to right (think of how calendars are laid out), so we think of causality the same way:

```
    this happened AND THEN that happened AND THEN the other happened
```

But our `silly` function goes the other way. The rightmost application (`times3` to `x`) must happen first, then the flow moves left. Elm provides a "pipeline" operator that lets us indulge our habits:

[5] Not a bad general rule, that. Especially in functional programming, which retains a cult of cleverness that's dying out in other types of programming.

```
> silly = times3 >> add2 >> minus8
<function> : number -> number
> silly 2
0 : number
```

`times3` is composed with `add2` to form a new function that in turn is composed with `minus8` to form a new function that's given the name `silly`. Here are the intermediate steps in the computation:

```
> times3 2
6 : number
> (times3 >> add2) 2
8 : number
> (times3 >> add2 >> minus8) 2
0 : number
```

Because the definition doesn't depend at all on the parameters – only on the original functions – it's written in point-free style. You actually could use a parameter in the definition, like this:

```
> silly x = (times3 >> add2 >> minus8) x
<function> : number -> number
```

... but that's so unconventional that it could only confuse an Elm programmer.

There is also a << operator that does the composition in the opposite direction. Consider the `List.filter` function:

```
> List.filter String.isEmpty ["", "foo", "", "bar"]
["",""] : List String
```

Selecting only the empty strings probably isn't very useful; we're more likely to want the *non*-empty strings. While we can do that using the >> operator:

```
> List.filter (String.isEmpty >> not) ["", "foo", "", "bar"]
["foo","bar"] : List String
```

... people used to languages that put negation before the verb might find that reads oddly, so Elm allows the following:

```
> List.filter (not << String.isEmpty) ["", "foo", "", "bar"]
["foo","bar"] : List String
```

As with the >> example, the `isEmpty` test is first and the result is "piped" to the `not`.

This operator is used less than >> is.

>> and << are most useful for combining a pipeline of functions into a single function that's then given to yet another function (like `List.filter`). In cases where you want to immediately apply the pipeline to a value, two other operators are more convenient. For example, consider this:

```
> (times3 >> add2 >> minus8) 2
0 : number
```

Awkward! The value we want to flow from left to right down the pipeline is given at the far right. The flow is clearer using the `|>` operator:

```
> 2 |> times3 |> add2 |> minus8
0 : number
```

As we saw with `Button.primaryButton`, `|>` combines well with partial application. Consider the following:

```
> 2 |> times3 |> (*) 2 |> minus8
4 : number
```

`(*) 2` is a partially-applied function that takes a single argument – which happens to be given by the previous step in the pipeline. Pipelines like that are used all over the place in Elm apps. Here's a typical example from some of my code:

```
model
  |> forgetHistoryPage history.id  -- produces a new version of `model`
  |> addCmd (Navigation.toPageChangeCmd next)
```

An important value (`model`) is passed as the last argument to two functions and transformed along the way. (Alternately, we could say that the model is passed as the only argument to two functions created by partial application.)

Notice that the `|>` saves you from having to add parentheses that would normally be required. The non-`|>` form of the above would need an extra set of parentheses:

```
addCmd
  (Navigation.toPageChangeCmd next)
  (forgetHistoryPage history.id model)
```

`|>` has a right-to-left variant: `<|`. I use it mainly as a replacement for parentheses. The following two are equivalent:

```
date = text (Date.humane entry.effectiveDate)
date = text <| Date.humane entry.effectiveDate
```

Which you prefer is entirely a matter of personal whim. I most often use '<|' to avoid nested parentheses. That is, I prefer this:

```
help title body =
  Just <| Css.dismissableModal title body (SetOverlay None)
```

… to this:

```
help title body =
  Just (Css.dismissableModal title body (SetOverlay None))
```

Reminder: there's nothing magic about `|>` and friends. They're functions like any other. You could (but never will) call `|>` as a function rather than use it as an operator:

```
> (|>) [1, 2, 3] (List.take 1)
[1] : List number
```

Exercises

1. Write a single `List.map` expression that takes a list of integers and produces a list of integers. Each integer is divided by 4 and then that result is negated (using `negate`).

   ```
   > List.map ???? [1, 5, 12]
   [-0.25,-1.25,-3] : List Float
   ```

   Hint in a footnote.[6]

   (Notice that Elm prints the integer `3` and floating point number `3.0` the same, so that final `-3` in the list really is a `Float`.)
2. Accomplish the same task with a `|>` pipeline:

[6]Use `flip` on `/` to convert "divide 4 by …" to "divide … by 4". It'll be easier if you fully parenthesize everything.

```
> [1, 5, 12] |> List.map ???? |> List.map ???
[-0.25,-1.25,-3] : List Float
```

3. For fun, reverse the << and |> operators in your solutions to the above two exercises to confirm that they really do make values flow in the other direction.
4. Your solution to the first problem should contain something like:

   ```
   flip (/) 4
   ```

   Can you write that subexpression using a |>? Now use your mini-pipeline inside solution. Do you find the result aesthetically pleasing?

Discussion and solutions

let

The body of every function is always a single expression. Given only what you've seen so far, that can get pretty ugly pretty fast, typically because of nested anonymous functions.

The `let` expression gives you a way to introduce temporary names. Here's a trivial example:

```
> let               \
|   first = 4 + 5   \
|   second = 6 - 8  \
| in                \
|   first * second
-18 : number
```

Notice that you tell the Elm repl to expect multiple lines by ending each with a backslash. If you're anything all like me, that's tremendously error-prone. In only two chapters, you'll be writing multi-line Elm functions in files but still trying them out in the repl.

Notice also the indentation. Elm will be unhappy if either the binding lines (after `let`) or the body (after `in`) aren't indented.

`let` is also used to define helper functions. Since functions are just values, you could bind an anonymous function to a name, but the function-naming shorthand also works:

```
> let                    \
|   add1 x = x + 1       \
| in                     \
|   List.map add1 [1, 2, 3]
[2,3,4] : List number
```

A bound name can be used both in other bindings (after the `let`) and in the body (after the `in`). That is, you can define a `add2` that referrs to `add1` (line 3):

```
1  > let                              \
2  |   add1 x = x + 1                 \
3  |   add2 x = x |> add1 |> add1     \
4  | in                               \
5  |   add2 43
6  45 : number
```

You can even define `add2` before `add1`:

```
> let                          \
|   add2 x = x |> add1 |> add1  \
|   add1 x = x + 1             \
| in                           \
|   add2 43
45 : number
```

The same free ordering is allowed, by the way, within a module. That allows you to arrange functions in a module/file in whatever way you think will be most informative for someone reading it from beginning to end.

Use-before-definition isn't true in the repl, though:

```
> add2 x = x |> add1 |> add1
-- NAMING ERROR ---------------------------

Cannot find variable `add1`

2| add2 x = x |> add1 |> add1
                 ^^^^
```

Two apparently pointless functions

Most functions do something useful. Some appear completely useless – but aren't. Here are two of them.

1. `identity` is a core Elm function that does nothing to its value:

```
> identity 3.5
3.5 : Float
```

2. `always` is another core function. It works like this:

   ```
   > (always "me") "I never get returned"
   "me" : String
   ```

   .. or:

   ```
   > meMeMe = always "me"
   <function> : b -> String

   > meMeMe "I never get returned"
   "me" : String
   ```

   That is, `always` takes an argument (like `"me"`) and produces a function that takes a single argument (such as `"I never get returned"`). That second function ignores its argument and instead always returns the argument to `always`.

Why should anyone care about these functions? You'll be working with many functions that apply some function that they're provided as an argument:

```
> calculateSomeValuesAndDoThisTothem negate moreArgs
```

How do you use such a function if you want the value left alone? – You pass in `identity`:

```
> calculateSomeValuesAndDoThisTothem identity moreArgs
```

What if you want the value ignored in favor of the number `3`? – You pass in `always 3`:

```
> calculateSomeValuesAndDoThisTothem (always 3) moreArgs
```

You'll see examples of such situations throughout the book.

Exercises

1. Implement `identity`.
2. Implement `always`.

   Hint in a footnote.[7]

Discussion and solutions

[7]An anonymous function that returns a function will have this structure: `\outer -> (\inner -> _____)`. Fill in the blank and give the function a name, preferably using the shorthand notation for functions.

What now?

In a book about statically *typed* functional programming, it seems natural to follow a chapter on functions with a chapter on types.

2. Some types

For now, we can think of types as nothing more than names used to lump together similar values. Just as you can say of an object in the physical world "*that* is a snuggly plush toy," you can say of some chunk of computer memory "*that* is a string."

The interesting thing is what happens when your statement is *wrong* – when the chunk of memory should actually be interpreted as an array of numbers, or what you think is a snuggly plush toy is actually an Ebola-riddled vervet monkey. There are several possibilities.

Both the physical world and low-level computer assembly languages are *untyped*. If you want to snuggle up to the vervet monkey as if it were inanimate, feel free! But the results may not be to your liking. When it comes to computers, trying to process a string that isn't a string could very well lead to your program stopping completely. That's bad if your program is controlling an airplane.

Languages like Javascript, Clojure, Elixir, Erlang, Ruby, Python, and Smalltalk aren't untyped. They're *dynamically typed*. That means that, at the moment you try to work with a "string," the language runtime knows whether it's truly a string or not. If not, you aren't allowed to touch it. What usually happens is that an exception is thrown. That can lead to the program stopping, but it doesn't have to. A more typical result, nowadays, is that the occasional user sees a confusing message about a "500 internal server error" in the browser. That's less drastic than a plane crash, and it may be acceptable.

In the vervet monkey analogy, dynamic typing would be like a mild electric shock that prevents you from getting too close to the monkey. That's better than getting Ebola from it, but you do still have this infectious monkey lying about.

A *statically typed* language like Elm or Purescript prevents you from fooling yourself about the monkey. You're not allowed to buy one while thinking it's a plush toy. If you want to mess with Ebola-laden monkeys, you have to affirmatively declare that's what you're going to do.

Similarly, in a statically typed language you simply are not allowed to execute a function that works with strings unless you can demonstrate to a gatekeeper (the compiler) that it is never possible for its arguments to be anything other than strings.

Nature? Untyped?

I just said the physical world is untyped. But aren't "tree," "cat," "bacteria," and the like types? I argue that they're a simplification imposed on reality. (But a useful one!) When you try to *really* find the clean boundaries between chairs and stools, trees and bushes, bachelors and non-bachelors, planets and dwarf planets, or different species of bacteria or birds, you find the situation can get very messy. Unlike `String` and `List` in Elm or Ruby, there's no boolean-valued function that distinguishes "chair" and "stool." Insisting too strongly on the abstraction is likely to cause problems, not benefits.

An interesting book on this topic is George Lakoff's *Women, Fire, and Dangerous Things: What Categories Reveal About the Mind.* I also recommend William James's *Pragmatism.* In fact, since William James predates Mickey Mouse,[1] I can reproduce his "story of the squirrel" in an appendix.

The old problem with static typing

All things being equal, it seems you'd always want to work in a statically-typed language. But all things aren't equal. The creators of statically-typed languages have struggled to make them as convenient to use as dynamically-typed languages. For example, languages like Java are notorious for making you tell the compiler things it already ought to know, over and over and over again. Fancy editors help, but do they help enough?

Languages like Elm and Purescript take a different approach. They make the compiler smarter so that it can *infer* many types. For example, consider a function `fnord` that inserts words in text:

```
> fnord "the third man"
"the fnord third fnord man" : String
```

Here's how you write it in Elm:

```
> fnord string = String.split " " string |> String.join " fnord "
```

I didn't have to declare that the `string` parameter in the definition of `fnord` is a `String`. Elm knows that it must be because `String.split` is applied to it, and `String.split` declares its second argument to be a `String`.

If I tried to compile an Elm program that applied `fnord` to a `Float`, the compiler would refuse to give you anything to execute, instead giving you a message like this:

[1]In the United States, the duration of a copyright keeps being extended whenever the first Mickey Mouse cartoon is about to go into the public domain.

```
-- TYPE MISMATCH ---------------------------------------------

The argument to function `fnord` is causing a mismatch.

3|   fnord 3.0
           ^^^
Function `fnord` is expecting the argument to be:

    String

But it is:

    Float
```

You would get the same message from the repl because it first compiles, then runs, every line you enter.

You can write a great deal of Elm code without having to declare any types at all. (Although, as we'll see, type declarations can be handy for documentation.)

There are other problems – and benefits! – to static typing. We'll talk about them throughout the book. For now, let's get concrete.

Concrete types

Elm has a variety of single-word types that begin with capital letters: `Bool`, `Int`, `Float`, `Char`, `String`, `Regex`, `Date`, `Time`, `Color`, and a few others. Here are some examples.

```
> "foo"
"foo" : String

> 1.0
1 : Float

> True
True : Bool

> Color.red
RGBA 204 0 0 1 : Color.Color
```

First Elm prints the value, then the type. `Color.Color` means the `Color` type in the module `Color`. It's common for a module to be "about" a type with the same name.

These types are examples of *concrete types*. We'll see more examples later.

Function types

Function types can be composed of concrete types. Here's the type of `fnord`:

```
> fnord
<function> : String -> String
```

`fnord` takes a string as its argument. The `-> String` shows that it produces a string. Simple enough. Things get more interesting when there's more than one argument, such as with `String.repeat`, which works like this:

```
> String.repeat 3 "--|"
"--|--|--|" : String
```

`String.repeat` takes two arguments – an integer and a string – and produces a new string. How are the types of the two arguments represented? Like this:

```
> String.repeat
<function> : Int -> String -> String
```

That's a bit peculiar. Why is the second argument represented by (`-> String`)? It's because of partial application. Remember that you can write code like this:

```
> repeat3 = String.repeat 3
> repeat3 "abc"
"abcabcabc" : String
```

What's the type of `repeat3` (which I sneakily omitted from the repl snippet above)? It's this:

```
> repeat3
<function> : String -> String
```

That makes sense: it takes a string and produces another string. So Elm's way of representing arguments is intended to remind you of partial application:

```
> String.repeat
<function> : Int -> String -> String
```

If you give a single integer argument, consuming the `Int ->`, you get a function that's `String -> String`. If you give a string argument to *that* function, the result is a `String`.

A way to understand this is to consider every function to take one argument. Therefore, what `String.repeat` *always* does is take an `Int` and produce a `String -> String` function. That's perhaps clearer if you parenthesize the arrows from the right:

```
> String.repeat
<function> : Int -> (String -> String)
```

… and you can complementarily parenthesize uses of `String.repeat`:

```
> (String.repeat 3) "abc"
```

The (unnecessarily) parenthesized expression produces a new single-argument function that's applied to `"abc"` to produce a `String`.

> If you'd like to know more about how partial application works, or you at some later point find yourself saying, "Wait - what's *really* going on here?", I've put more details in an appendix: "The singular truth about functions."

Quantified types

Lists are more interesting than strings or numbers. Elm insists that a list be a list *of* something - that is, that all of its elements have to be of a particular type. So you can have two distinct list types:

```
> [1.0, 2.0, 3.0]
[1,2,3] : List Float

> ["a", "b", "c"]
["a","b","c"] : List String
```

Elm has a function, `List.length`. It works on a list of any type and produces an `Int`. Here's that function's type:

```
> List.length
<function> : List a -> Int
```

The lowercase `a` is a *type variable*. It marks a place where a type must be present, but it doesn't say anything about what type that should be.

Here's another function that works on two lists. It's not restricted to, say, `List Float` or `List String`:

```
> List.append [1.0, 2.0] [3.0, 4.0]
[1,2,3,4] : List Float

> List.append ["a"] ["b", "c"]
["a","b","c"] : List String
```

However, it does place one restriction on its arguments: they must be lists of the same concrete type. Trying to append strings to floats, like this:

```
> List.append [1.0, 2.0, 3.0] ["a", "b", "c"]
```

... produces this:

```
-- TYPE MISMATCH ---------------------------------------------

The 2nd argument to function `append` is causing a mismatch.

3|   List.append [1.0, 2.0, 3.0] ["a", "b", "c"]
                                 ^^^^^^^^^^^^^^^
Function `append` is expecting the 2nd argument to be:

    List Float

But it is:

    List String
```

That restriction is written like this:

```
> List.append
<function> : List a -> List a -> List a
```

It's the repetition of the `a` that tells you that the element types of the two lists must be the same. If the first is a `Float`, so must the second be.

Let's see what happens when we partially apply `List.append`:

```
> List.append [1.0, 2.0]
<function> : List Float -> List Float
```

The compiler knows three things about `List.append`:

1. Its first argument can be any type of list.
2. The second argument must be the *same* type of list.
3. And so will be the return type.

So once the first argument's type is fixed by giving it a *particular* list, the compiler knows the the other argument and the result must have the same type.

`List.append` is a fine function, but there are functions of two lists that shouldn't care about element types. For example, consider a function that adds the lengths of two lists:

```
> length2 [1, 2, 3] ["a", "b", "c"]
6 : Int
```

Here's a definition for `length2`, together with the type Elm infers for it:

```
> length2 list1 list2 = List.length list1 + List.length list2
<function> : List a -> List a1 -> Int
```

Elm indicates the two lists may be of different type by using different type variables: `a` and `a1`.

If we fixed the type of the first argument, the second argument could still be any type of list.

```
> length2 [1.0]
<function> : List a -> Int
```

Names

Quantified types are types with type variables. They can also be called *abstract types*. Whatever the name, it's the type variables that distinguish such types from concrete types.

I'll now reveal that concrete types aren't just single-word types like `Int`, but rather any type without type variables. So:

```
List (List a)        -- abstract type
List (List Float)    -- Concrete type, values like [ [ 1.3 ] ]
```

Elm cheats, a bit

Elm has three special type variables: `number`, `comparable`, and `appendable`. Unlike other type variables, they aren't blanks into which you can write any concrete type. They only allow a few possibilities.

That is, when you see declarations like this:

```
negate : number -> number
```

... or this:

```
max : comparable -> comparable -> comparable
```

... or this:

```
> (++)
<function> : appendable -> appendable -> appendable
```

... you should know that's different than declarations that use type variables like `a`, `result`, or `firstArgType`. You've encountered a special case.

`number`

Arithmetic is annoying to programming language designers because we programmers picked up certain habits early in school, habits that don't really translate well into the stricter world of the computer. Since we usually refuse to give up our habits, languages have to adapt.

For example, if you want to be picky about it, `Float` and `Int` are different types. Therefore, the function to add two integers should have one type:

```
addInts : Int -> Int -> Int
```

… and the function for adding floats should have another:

```
addFloats : Float -> Float -> Float
```

… and let’s not forget about adding integers and floats:

```
addIntAndFloat : Int -> Float -> Float
addFloatAndInt : Float -> Int -> Float
```

That would be really annoying. There are many ways to handle such a problem. Elm chooses a simple one: there’s a special type variable `number` that allows either (but only) `Float` or `Int`.

comparable

`comparable` solves a similar problem. We want `(>)` to work with both integers and floats. And it’s also convenient if it works with lists, provided the elements are of the same type:

```
> [1, 2, 3] > [1, 2, 2]
True : Bool
```

If characters are convertible to integers (which they are), you should also be able to compare both characters and strings:

```
> "ZZZ" > "AAA"
True : Bool
> "aaa" > "ZZZ"
True : Bool
```

(It’s really annoying that “aaa” > “ZZZ” > “AAA,” but that’s the world we’ve inherited.)

The Elm documentation has a list of all `comparable` types.

appendable

`appendable` lets `(++)` work on both strings and lists:

```
> "abc" ++ "def"
"abcdef" : String
> ["abc"] ++ ["de", "f"]
["abc","de","f"] : List String
```

Functions that take functions as arguments

When functions take functions as arguments, the type of the function argument is surrounded with parentheses. Here's `List.map`:

```
> List.map
<function> : (aaa -> bbb) -> List aaa -> List bbb
```

(What Elm actually prints uses the single letters `a` and `b`, but it's too jarring to read sentences like "it takes an `a` and produces a `b`," so I'm overruling Elm.)

That type shows that the first parameter must be another function. It takes an `aaa` and produces a `bbb`. Given what `map` does – feed elements from its second argument to its first – the second parameter's type variable has to be `aaa`. Similarly, the type variable in the return type must be `bbb`.

As we saw earlier, partial application can force other types to become more specific. For example:

```
> List.map String.length
<function> : List String -> List Int
```

From this you can deduce that `String.length` must take a `String` and produce an `Int`.

Note: It's perfectly fine to use the same concrete type for both `aaa` and `bbb` (or, generally, for any distinct type variables). For example:

```
> List.map String.toUpper
<function> : List String -> List String
```

Both `aaa` and `bbb` have been replaced with `String`.

What now?

This chapter was about how to read types Elm provides. The next is about how you write types for Elm. You also start getting to work with files.

3. Type annotations

You don't have to rely on Elm to tell you the type of a function or other value. You can write it down yourself before compiling.

Until now, we've done everything in the repl. But that's awkward for multi-line examples – at least if you make as many typos as I do and are as bad at noticing them until it's too late. Moreover, some code can't be written in the repl. Like the code this chapter is about.

The book's code repository has a file, elm/Scratch.elm that you can use for examples. It describes how you can write a function or code snippet there and quickly use it in the repl.

Terminology. Yay. Terminology

I'm not a stickler for terminology, but a little precision will keep things clear going forward. Consider the following function and the type Elm infers (which I'll call the *inferred type* from now on).

```
-- In Scratch.elm:
length2 list1 list2 =
  List.length list1 + List.length list2

-- In the repl:
> import Scratch exposing (..)
> length2
<function> : List a -> List a1 -> Int
```

I can add a matching *type annotation* just before the function definition. I don't much care for the type variables `a` and `a1`, but I don't have any particularly descriptive names (because it's kind of a dumb function), so I'll just use `a` and `b`:

```
length2 : List a -> List b -> Int
length2 list1 list2 =
  List.length list1 + List.length list2
```

The repl will show your annotation instead of the inferred type:

```
-- Must re-import. Compilation is automatic.
> import Scratch exposing (..)
> length2
<function> : List a -> List b -> Int
```

Plain values can also have type annotations:

```
names : List String
names = ["Dawn", "Paul", "Sophie"]
```

Elm is still in charge

Whether or not you provide an annotation, Elm calculates the inferred type. Your type annotation may not contradict Elm. Here's a function and its inferred type:

```
addFive list =
  List.append list [5.0]

> addFive
<function> : List Float -> List Float
```

You cannot add a more forgiving, "broader" annotation like this one:

```
addFive : List anything -> List anything
```

That causes an error:

```
The 2nd argument to function `append` is causing a mismatch.

65|   List.append list [5.0]
                       ^^^^^
Function `append` is expecting the 2nd argument to be:

    List anything

But it is:

    List Float

Hint: Your type annotation uses type variable `anything`
```

```
which means any type of value can flow through.
Your code is saying it CANNOT be anything though!
Maybe change your type annotation to be more specific?
Maybe the code has a problem?
```

When it's irrelevant whether the type of a function comes from an annotation or was inferred by Elm, I'll call it the function's *type signature*.

Exercise

1. The file `TypeAnnotations/Errors.elm` has functions with incorrect type annotations. An easy way to see them is to type this in while in the `elm` directory:

   ```
   elm-make TypeAnnotations/Errors.elm
   ```

   Fix the problems.

Discussion and solutions

Narrowing

While your type annotation can't be broader than Elm's inferred type, it can be more specific. That's usually done by replacing a type variable with a concrete type. Here's a function and its type:

```
negatory thing =
  negate thing

> negatory
<function> : number -> number
```

If I wanted to allow `negatory` to be used only with `Float` values, I'd use this type signature:

```
negatory : Float -> Float
negatory thing =
  negate thing

> negatory
<function> : Float -> Float
```

Now Elm will object if you try to compile a program that applies `negatory` to an `Int`.

A compiler can use a narrowed type annotation to produce better (faster, more specific) code because it knows more about the possible inputs. I'm going to ignore that in this book: for us, a type annotation is about preventing misuse, providing better documentation, and allowing some clever solutions (described later) to common problems.

What now?

Elm doesn't require type annotations, but I do recommend you use them throughout this book. In particular, start creating each function by writing its annotation. If you don't want to do that, Elm provides a tool (`elm-make --warn`) that notices if a function doesn't have an annotation and suggests one to you:

```
1138 $ elm-make --warn Scratch.elm
=================================== WARNINGS ===================================

-- missing type annotation ----------------------------------------- Scratch.elm

Top-level value `length2` does not have a type annotation.

8| length2 list1 list2 =
   ^^^^^^^
I inferred the type annotation so you can copy it into your code:

length2 : List a -> List a1 -> Int
```

At this point, I want to dive into a quantified type, `Maybe a`, and introduce sum types, which are very nice. However, I've been leaving unsaid something that you should understand: in functional programming, you don't change data. Instead, you create changed copies.

4. A digression: immutability

Most any discussion of functional programming will make a big deal of “immutability.” I personally think it’s much less life-changing than partial application or sum types (coming up!), but you need to understand it to be a functional programmer.

I’ll explain it using `List` as an example, along the way showing you a useful operator.

I’ll also explain some common jargon: “pure functions” and “referential transparency.” They and immutability are all interrelated. Since these languages give you all three at once, you don’t actually need to be able to make the distinctions. You just need to know what you can’t do in these languages and what you do instead.

Building lists

Here’s an empty list:

```
> []
[] : List a
```

Because there are no elements, the type variable can’t be assigned a concrete type, so it’s left alone.

That means we can add anything we want to the empty list. There’s an operator, `(::)`, for doing that:

```
> first = "first" :: []
["first"] : List String
```

We now have a list of strings, and we don’t have a choice about what else we can add to that. Anything but a string will produce an error. This:

```
> 3.5 :: first
```

… produces this:

```
-- TYPE MISMATCH ------------------------------------------- repl-temp-000.elm

The right side of (::) is causing a type mismatch.

3|   3.5 :: first
            ^^^^^
(::) is expecting the right side to be a:

    List Float

But the right side is:

    List String
```

Notice that `::` adds to the front of a list:

```
> "second" :: ["first"]
["second","first"] : List String
```

For historical reasons, `::` is pronounced “cons” (with a “zee” sound at the end). Above we “consed” a string onto a list.

Eu sou verde, e daí?[1]

Although I talked about “adding to the list,” the common English meaning of the word “add” does *not* describe what’s happening. When you add to a list, the list is not changed. Consider this code:

[1]Dawn and I are learning Portuguese, and one of our study tools is Duolingo. It has a habit of throwing up sentences that are… not likely to come up in normal conversation, and “Eu sou verde, e daí?” is our favorite. It means “I am green, so what?” I didn’t want a section named “Immutability” in a chapter named “Immutability,” and this favorite sentence seems a match for my rather flippant attitude toward the **serious business** of immutability. (As in the introduction, I have to add “Schrödinger’s” to get special characters to work. This is bafflingly inconsistent.)

```
> zero = []
[] : List a
> one = 1.1 :: zero
[1.1] : List Float
> two = 2.2 :: one
[2.2,1.1] : List Float
```

Each line has **no effect** on the results of the previous line. The names point to unchanged values:

```
> zero
[] : List a
> one
[1.1] : List Float
> two
[2.2,1.1] : List Float
```

Conceptually, every one of the `::` operations created an entirely new list. (In reality, structure is shared behind the scenes, but there's no way for your Elm code to detect that.)

Lists aren't special. There is no function in Elm that modifies (or *mutates*) data. You'll find functions named `insert`, `append`, `concat`, and so on. In every case, you should think of them as creating entirely new structures:

```
> List.append one two
[1.1,2.2,1.1] : List Float
> one
[1.1] : List Float
> two
[2.2,1.1] : List Float
```

The benefit of immutability is that it prevents bugs that happen when one piece of code changes a bit of data that another piece of code expected to stay the same. This is a *huge* benefit when it comes to multithread programming. I personally think it's a mild benefit when it comes to ordinary programming. That is, it does prevent some kinds of bugs, but other kinds of bugs are usually a lot more important.

My opinion is unpopular. When travelling in FP circles, people will assure you "mutable code is too hard to reason about." I suggest you smile and nod politely.

People have two negative reactions to immutability:

1. "It's incredibly inefficient!" There are two answers. The first – especially appealing to me, whose first job was programming a computer with 65,536 words of data space – is:

"Who cares?" We have lots of memory, garbage collectors have gotten pretty good, so most programmers don't have to worry about such things.

While that's true, it's also worth remembering that the language runtime can mutate all it wants, so long as no code can tell. So, in FP languages whose designers obsess over efficiency (Clojure is a good example), changing the 8000th element in a huge vector doesn't produce another huge vector. It really produces a small structure that says "this is a vector whose 8000th element is 5; all of its other elements are the same as that vector over there."

2. "I will find it really hard to write code without `a.foo = 3` and the like." My honest answer is: it's not hard once you're used to it. For most everything you want to do, someone has already come up with a clever alternative. For example, here's Ruby code that updates a field within a structure within a structure:

   ```
   model.effective_date.picker_state = CLOSED
   ```

   Immutability makes that awkward because you have to do the equivalent of the following (where `dup` is how you copy something in Ruby):

   ```
   new_effective_date = model.effective_date.dup
   new_effective_date.picker_state = CLOSED
   new_model = model.dup
   new_model.effective_date = new_effective_date
   new_model
   ```

   However, *lenses* (which you'll see – and implement! – in a later chapter) make this considerably less annoying. It's still somewhat annoying, but let's put that in perspective. It's *much less* annoying than the FP languages' insistence on `UseOfStudlyCapsForNames` instead of the `infinitely_more_readable_underscore_style`. (This is the point where you smile and nod politely at me.)

So, the upshot is: immutability is part of FP. It's no big deal.

"Purity" and other jargon

A pure function is one with two properties:

1. Whenever you apply the function to the same arguments, you get the same result. Some functions you probably use without a second thought aren't pure. They have names like `Time.now`, `random`, `read`, and `Http.get`.
2. The function can make no observable change to any state. What follows is a Ruby function that's not pure because it changes a global variable `$number_of_sums`:

```
def sum(x, y)
  $number_of_sums ||= 0      # lazily initialize
  $number_of_sums += 1
  x + y
end
```

Similarly, a pure function can't change the contents of a file.

You may also hear the phrase "referential transparency." What that means is you can always substitute the value of an expression for the expression itself. Pure functions are always referentially transparent. What that means is that, anywhere you see this function application:

```
negate 3
```

... you can replace it with `-3`. More generally, referential transparency means that evaluating an Elm program to get a value requires nothing more than simple substitution. Consider this function:

```
double x = x + x
```

... and remember that's really just shorthand for this definition:

```
double = \x -> x + x
```

Referential transparency means that this:

```
(double 3) + (double 2)
```

... must produce the same value as this:

```
((\x -> x + x) 3) + ((\x -> x + x) 2)
```

... because we substituted the value of `double` for the name `double`.

We can continue creating expressions that can be freely substituted for the original.

Applying the first function means substituting `3` for `x` in `(x + x)`. The same thing is done with `2`. So the previous expression must have the same value as this:

```
(3 + 3) + (2 + 2)
```

... and the same value as this:

```
6 + 4
```

… and this:

```
10
```

Substitution is a nice, simple process. Calculating the result of a program is considerably more complicated when you have to worry about global variables, assignment statements, and other things that break purity and referential transparency. However, that's not actually our problem as programmers – it's the compiler's problem.[2]

All of our languages give you immutability, referential transparency, and pure functions. So you don't need to remember the distinctions.

Practical implications: the outside world

When you write code in these languages, you create pure functions that can't do input or output. You combine those pure functions into larger functions, eventually into a single "main" function. Such compositions are themselves pure, which means your entire program is pure. Which means your program can't do input or output.

Wait. What?

It would be somewhat… disappointing if all of our programs were, in effect, run in the repl… if they were just larger versions of `double`:

```
> double 3
6 : number
```

… where we provide all the data as arguments to the "main" function, get answers printed at us, then – I dunno – copy the result and use SQL to put it in a Postgres database so it can be read by our app server.

So there's a need to live within the letter of the immutability-and-friends law while still being able to write useful programs. Here's a simplified description of how Elm does that:

1. Your program does nothing but consume data and produce data. It does no I/O.

[2]Some argue that what makes the compiler's problem harder also makes it harder for programmers to understand what the code does. That's probably true, but I'm agnostic on how much it actually matters. A compiler has to handle *any* correct code you give it, but as people we expect to read code written by sane programmers with some concern for the understandability of what they write. You are no doubt this very instant picturing that one programmer whose code was anything *but* sane. However, I have this sneaking suspicion that forcing referential transparency on him would fix only a small part of the problem.

2. The Elm runtime will call your program many times, each time giving it two arguments: the state of the application (“the model”) and a description of some external event (“a message”).
3. Your program returns a pair: a new model and some data (“a command”) that requests the runtime do something for you. (The command can be `Cmd.none`, in which case the runtime waits around for some external event, like a button click.)
4. Your program remains pure. It’s the runtime that replaces the old model with the new model and makes changes to the outside world.

That works nicely for things like sending HTTP requests. It avoids “callback hell” by decoupling the sending of a request from the processing of a response. The request is sent in one call to your program, and the response is handled in a later one. Your program needn’t remember it ever made a request; it can treat an HTTP response as an unprovoked event no different than a button press. (Or it can store some memory in the model.)

However. (There’s always a however.) While this approach, called the *Elm Architecture*, makes the complex simple, it can make the simple awkward. For example, one of my programs needs the current date. I get it like this:

1. During one call, my program returns this command to the runtime:

   ```
   Task.perform SetToday Date.now
   ```

   `SetToday` is a message (that I had to define) that will hold the resulting datetime.
2. The runtime knows that the command `Date.now` means that it should call the JavaScript function `Date.now` and package the result in a `SetToday` message.
3. Quite soon after that, my program is invoked again with the model and the message. It puts the current date in (an immutable copy of) the model.

This is rather involved compared to Ruby:

```
model.current_date = Date.today
```

Moreover, in this model, sequences of interaction like this Ruby code:

```
puts "Type something"
x = gets
puts "You typed this: ", x
```

… can become awkward because it’s you that must impose the ordering in time, not the language or runtime.

At the cost of some conceptual complexity, Purescript and other languages allow sequences that *look* like the above:

```
cmd = do
  putStrLn "Type something"
  x <- readLine
  putStrLn "You typed this: " ++ x
```

However, what's really happening is that the different I/O commands are assembled into a single aggregate command. As with Elm, "someone else" does the dirty work of executing it. What's changed is that you're given a more powerful "domain-specific language" for instructing that someone else and organizing how your non-I/O code is used.

In the immortal words of the last line of the only block comment in the Version 6 Unix kernel: "you are not expected to understand this." But you will!

What now?

Besides partial application, *sum types* are my favorite thing about static FP. I'm going to take two chapters to describe them. The first will be all about a particular sum type: `Maybe t`. Because `Maybe` is enormously heavily used, that will give us a break from concept after concept after concept – I'll be able to describe some idioms.

5. Maybe and its idioms

In the previous chapter, I showed you how to “add” an element to the front of an array:

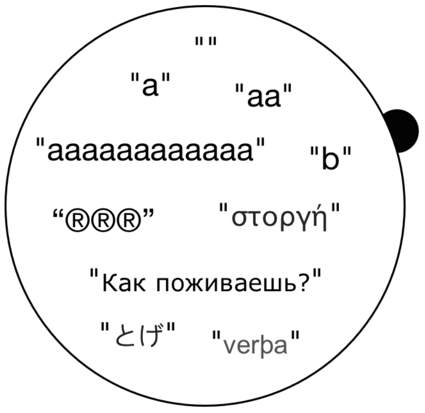
```
> 1 :: []
[1] : List number
```

How do you retrieve it, once it’s there? You use a function called `List.head`. I won’t show you an example yet because there’s a special case: what happens if the list is empty?

In many languages, there’s a simple solution: return `null`, or `nil`, or some other word that conveys the idea “nothing here, boss.” Here’s how Clojure handles it:

```
user=> (first [])
nil
```

In such a scheme, a type like `String` *really* means “the set of all possible strings, plus this… *thing* on the side that you should remember is a special case.”



A Type With a Blemish

We might write a definition of this type like the following, where `|` means “or”:

```
type String
    = TrueString
    | Null
```

There are some potential problems here. A given null-containing language might not have some or any of these problems, but they tend to.

1. There's nothing that forces you to remember a value might be `Null`, so you might not check for it.
2. You only get the name `String`, not `TrueString`, so there's no way to designate a function or variable as not "nullable" and have the compiler enforce it.
3. Even though many, many types include the null-blemish, there's no overarching category of "types that are sometimes null." That makes it unnatural or impossible to write code that abstracts away that detail.

The static FP languages use the type system to address problems 2 and 3.

- They use type variables to create an overarching type. All of our languages call it `Maybe`:

  ```
  type Maybe a ...
  ```

  So you can create a `Maybe String`, a `Maybe Bool`, a `Maybe Float`, etc., etc.
- There's an explicit representation for a value that could have been `null`, but was actually legitimate. For a string, that representation is this:

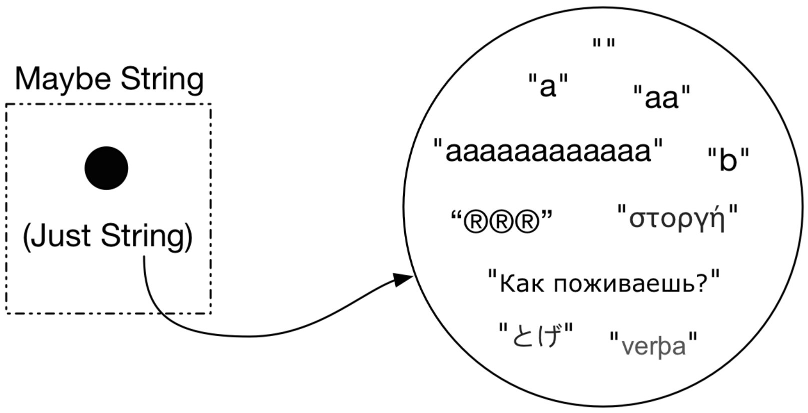
  ```
  Just "string result"
  ```

  (I would have chosen `Actual`, as in `Actual "string result"`, but no one asked me.)
- The other possibility (no actual value) is typically named `Nothing`.

Putting that all together, we have this type, straight from the Elm source:

```
type Maybe a
    = Just a
    | Nothing
```

I think of it like this:



Maybe

That’s our first example of a *sum type*. It’s a type that represents two or more different kinds of values that cannot be mixed. It’s called a “sum” type because the number of elements of a type like `Maybe Float` is the sum of the number of possible floating-point values (a large but finite number) plus one (the single value `Nothing`). Elm and Idris programmers tend to use the term *union type* instead, though.

You’ll see more sum types in the next chapter. For now, let’s see what we can do with this particular one.

About type variables and words

It’s conventional for functional programmers to use single letters for type variables, as with `Maybe a`. I think they overdo it, sticking to that convention when more descriptive names would be better.

I was tempted to flout the convention, but I eventually decided to (mostly) obey it. I want what I show to match what you’d see in the repl and documentation.

However, that leaves the ugliness of sentences like “`Maybe a` is a…”. They’re jarring because `a` and the word “a” are not visually distinct enough. In English sentences, therefore, I’ll tend to enclose such awkward types and types variables in parentheses: “`(MyType a b)` has two type variables: `(a)` and `(b)`.” Those parentheses are valid in Elm code, too, but they’re not needed there.

Making use of `Maybe`

Back to `List.head`. Here’s its type:

```
> List.head
<function> : List a -> Maybe.Maybe a
```

(Notice that the type (Maybe a) is defined in the module Maybe.)

And here's how it's used:

```
> List.head ["Dawn"]
Just "Dawn" : Maybe.Maybe String
> List.head []
Nothing : Maybe.Maybe a
```

Elm does not allow you to make the mistake of treating a Just String like a String. This:

```
> String.length (List.head ["Dawn"])
```

... produces this:

```
-- TYPE MISMATCH -------------------------------------------- repl-temp-000.elm

The argument to function `length` is causing a mismatch.

3|   String.length (List.head ["Dawn"])
                    ^^^^^^^^^^^^^^^^^^
Function `length` is expecting the argument to be:

    String

But it is:

    Maybe String
```

For now, think of a String like "Dawn" as being "wrapped" by Just. (The reality is more abstract, but we won't have to worry about that for many pages.) Before you can work on it as a String, you have to unwrap it. The basic way you do that is with a case expression, which looks like this:

```
case List.head ["Dawn"] of
  Just name ->
    String.length name
```

This `case` expression is an example of *pattern matching*. Here's what's matched:

```
Just "Dawn" -- value of `List.head ["Dawn"]`
Just name   -- pattern
```

That match succeeds. Because it does, the variable `name` becomes bound to the unwrapped value `"Dawn"`. That variable can be used for computation in code after the `case`'s `->` (but not outside it).

You'll see more of pattern matching later. It's one of the nice things about static FP languages, though it's not limited to them. Elixir, for example, makes good use of it.

Pattern matching practice

For the following value/pattern pairs, determine if the pattern matches. If so, what value is `variable` bound to?

1. Value `Just "Dawn"` and pattern `(Just variable)`
2. Value `Just "Dawn"` and pattern `Just "Brian"`
3. Value `Just "Dawn"` and pattern `Nothing`
4. Value `Just "Dawn"` and pattern `Ok "Brian"`
5. Value `Just "Dawn"` and pattern `Ok variable`
6. Value `Just (Just "Dawn")` and pattern `Just (Just variable)`
7. Value `Just (Just "Dawn")` and pattern `Just variable`

Discussion and solutions

Let's return again to our case expression:

```
case List.head ["Dawn"] of
  Just name ->
    String.length name
```

It's not finished yet. If you try to use it, you'll see this:

```
-- MISSING PATTERNS ----------------------------------------- repl-temp-000.elm

This `case` does not have branches for all possibilities.

3|>  case List.head ["Dawn"] of
4|>    Just name ->
5|>      String.length name

You need to account for the following values:

    Maybe.Nothing

Add a branch to cover this pattern!
```

In Elm, a `case` must have a clause for each possibility. That is, we must fill in the blank here:

```
case List.head ["Dawn"] of
  Just name ->
    String.length name
  Nothing ->
    __________________ -- Profit?
```

It's often difficult to think of something useful to do in the `Nothing` case. For now, let's just say that the length of `Nothing` is `0`, and get this:

```
> case List.head ["Dawn"] of      \
    Just name ->                  \
      String.length name          \
    Nothing ->                    \
      0
4 : Int
```

There. That's `Maybe`. I'm hoping you are feeling two emotions right now:

1. You should be **afraid** that your code will be littered with `case` expressions: error handling that makes the normal case too difficult to see.
2. You should be **annoyed** that you'll be spending way too much time coming up with bogus `Nothing` cases like "the length of no string is zero."

That's because you've only seen an explanation of `Maybe`. You haven't seen the idioms for it. That's coming up next. But first…

Exercises

I suggest you edit the file `elm/Maybe/One.elm` in the book's source. That file contains some helpful commentary.

1. Elm has `True` and `False` values. You'll see more about them in the next chapter. For now, write a function `toBool` that takes a `(Maybe a)` value and produces either `True` or `False`.

   ```
   > One.toBool (Just 3)
   True : Bool

   > One.toBool Nothing
   False : Bool
   ```

   Try writing the type annotation before you write the function. `True` and `False` are values of type `Bool`.
2. Sometimes you can end up with nested `Maybe` values, like `Just (Just 3)` or `Just Nothing`. Write a function `join` that removes the nesting, as in these examples:

   ```
   > One.join Nothing
   Nothing : Maybe.Maybe a

   > One.join <| Just Nothing
   Nothing : Maybe.Maybe a

   > One.join <| Just (Just 3)
   Just 3 : Maybe.Maybe number
   ```

   Again, write the type annotation first.

   What do you get if you apply your function like this?

   ```
   > One.join (Just 3)
   ```
3. Write a function `toList` that converts a `Maybe a` into a `List a`:

   ```
   > One.toList Nothing
   [] : List a

   > One.toList (Just 3)
   [3] : List number
   ```

Discussion and solutions

Maybe.map and pipelines of iffy computations

The String module has many functions that operate on strings: isEmpty, length, reverse, append, split, and on and on. Now that we have Maybe, we certainly don't want to copy, paste, and create another bunch of functions that operate on Maybe String. It'd be boring, trivial work, the sort we want computers to do.

Instead, we want someone to give us *one* additional function that does the conversion for us. In the FP jargon, this is called *lifting*. Metaphorically, you are "lifting" a function that operates in a simple universe into a corresponding function for a more complicated universe (in this case, one with a weird additional value called Nothing).

The name of that one additional function is Maybe.map. Here's its definition:

```
map : (arg -> result) -> Maybe arg -> Maybe result
map f maybe =
    case maybe of
        Nothing -> Nothing
        Just arg -> Just (f arg)
```

In the weird Nothing case, the result of "mapping a function over" the argument is… Nothing. In the immortal words of Billy Preston, "Nothing from nothing leaves nothing / You gotta have something if you want to be with me." Or, as an example:

```
> Maybe.map String.reverse Nothing
Nothing : Maybe.Maybe String
```

When there *is* something (that is, a value wrapped by Just), the function is applied to the unwrapped value, and the result is rewrapped:

```
> Maybe.map String.reverse (Just "Dawn")
Just "nwaD" : Maybe.Maybe String
```

Maybe.map's behavior with Nothing means that you can pipeline functions together without any of them having to worry about the Nothing value:

```
> Nothing
    |> Maybe.map String.reverse
    |> Maybe.map (String.append "Dawn")
Nothing : Maybe.Maybe String

> Just "Dawn"
    |> Maybe.map String.reverse
    |> Maybe.map (String.append "Dawn")
Just "DawnnwaD" : Maybe.Maybe String
```

That's handy for functions lifted from the "nothing-free universe," but not so handy for functions that themselves return a `Maybe`. Here's an oddly-named one: `String.uncons`. It works like this:

```
> String.uncons "Dawn"
Just ('D',"awn") : Maybe.Maybe ( Char, String )
```

Let's take apart `('D', "awn")`. `"awn"` is a string, which you've seen before. The single quotes around `'D'` mean the value is a single character. So `uncons` has separated the first character from the rest of the string and returned both in a *tuple*. Tuples are a bit like lists, but you write them using parentheses instead of square brackets:

```
> [1.3, 2.6]
[1,2] : List Float

> (1.3, 2.6)
(1.3,2.6) : ( Float, Float )
```

Unlike lists, tuples can contain different types:

```
> (1.3, "Dawn")
(1.3,"Dawn") : ( Float, String )
```

But tuples can't grow. There's no tuple equivalent to this:

```
> 8.8 :: [1.3, 2.6]
[8.8,1.3,2.6] : List Float
```

You'll be seeing more of tuples later.

Meanwhile, `String.uncons` doesn't just return a tuple. Because an empty string has no first character, it must return a tuple wrapped in a `Maybe`:

```
> String.uncons ""
Nothing : Maybe.Maybe ( Char, String )

> String.uncons "a"
Just ('a',"") : Maybe.Maybe ( Char, String )
```

For fun, let's map `String.uncons` over a `Maybe` value. (That is, use a `Maybe` as input instead of just expecting one as output.) I'll spread the type of the result over several lines to make it clearer what's going on:

```
> Just "Dawn" |> Maybe.map String.uncons
Just (Just ('D',"awn")) :
   Maybe.Maybe
      (Maybe.Maybe
         ( Char, String ))
```

What's happened here?

1. `Maybe.map` extracts the value `"Dawn"` and gives it to `String.uncons`.
2. Since `"Dawn"` isn't empty, `String.uncons` produces `Just ('D',"awn")`.
3. Per its definition, `Maybe.map` wraps the result in (another!) `Just`.

Let's avoid double-wrapping with a function that's an alternative to `map`. It's called `andThen`. The reasoning behind it goes like this:

1. `Maybe.map` handles functions that take "our universe" values and convert them to "our universe" values. `negate` is such a function: it takes numbers and produces numbers. It's entirely innocent of the existence of `Maybe`.
2. But there are some functions that take "our universe" values and produce `Maybe` universe values (`Just` or `Nothing`).
3. `andThen` should expect that and not (as `map` does) wrap the function's result in a `Just`. Instead, it should just return it.

Here's the type of `andThen`, set alongside that of `map`, with spacing added so you can more easily compare the two:

```
map     : (arg ->       result) -> Maybe arg -> Maybe result
andThen : (arg -> Maybe result) -> Maybe arg -> Maybe result
```

Its implementation differs from `map`'s only in that it doesn't wrap the function's result in `Just`. See line 4 below:

```
map f maybe =                  |   andThen f maybe =
  case maybe of                |     case maybe of
    Nothing -> Nothing         |       Nothing -> Nothing
    Just arg -> Just (f arg)   |       Just arg -> f arg
```

To show the use of `map` and `andThen`, let me define a function `headButLast` that takes the `head` of a `List String` and removes its last character:

```
> headButLast ["Dawn"]
Just "Daw" : Maybe.Maybe String
```

Such a function must handle two unhappy cases:

1. The list is empty:

   ```
   > headButLast []
   Nothing : Maybe.Maybe String
   ```

2. The list has a `head` but it's the empty string (and the empty string doesn't have a last character):

   ```
   > headButLast [""]
   Nothing : Maybe.Maybe String
   ```

Here's the implementation:

```
headButLast : List String -> Maybe String
headButLast list =
  list                                -- ["Dawn"]
    |> List.head                      -- Just "Dawn"
    |> Maybe.map String.reverse       -- Just "nwaD"
    |> Maybe.andThen String.uncons    -- Just (n, "waD")
    |> Maybe.map Tuple.second         -- Just "waD"
    |> Maybe.map String.reverse       -- Just "Daw"
```

I think this is pretty cool. No `case` expressions anywhere. But what's actually going on?

1. If `List.head` produces `Nothing`, all the following `Maybe.map` and `Maybe.andThen` lines will just forward on the `Nothing`, making the final result `Nothing`.
2. Otherwise, if the original list was `[""]`, `String.uncons` will produce `Nothing`. And that means the following uses of `Maybe.map` won't actually apply their function arguments but rather just forward `Nothing` on.
3. Only in the case that those two hurdles are passed does the final `Maybe.map` produce a `Just`.

Note that the final result is still in the `Maybe` universe. We'll handle that next.

Exercise

Create a pipeline that takes a string and produces its second character, wrapped in `Maybe`. Its type signature should be this:

```
secondCharacter : String -> Maybe Char
```

Here are some test cases:

```
> import Maybe.TwoSolution as Two
> Two.secondCharacter ""
Nothing : Maybe.Maybe Char

> Two.secondCharacter "a"
Nothing : Maybe.Maybe Char

> Two.secondCharacter "ab"
Just 'b' : Maybe.Maybe Char

> Two.secondCharacter "abc"
Just 'b' : Maybe.Maybe Char
```

Hint in a footnote.[1]

Discussion and solutions

At the end of the pipeline

If you're just printing a result to the repl, letting the pipeline return a `Maybe` is fine. But if your purpose was to produce a value that has some effect on the outside world – say, it's to be converted into JSON – you have to move that value out of the `Maybe` universe, and you have to deal with the possibility that there may be no value there (it's `Nothing`).

The function to use is `Maybe.withDefault`. It takes a default value and a `Maybe`:

```
> Maybe.withDefault
<function> : a -> Maybe.Maybe a -> a
```

Its return value is either the wrapped `Just` value or the default:

[1]Use `String.uncons` twice in the pipeline.

```
> Maybe.withDefault 0 (Just 4)
4 : number
> Maybe.withDefault 0 Nothing
0 : number
```

Notice that `withDefault` follows the idiom of putting the most important value last, so it fits well at the end of a pipeline. Recall our earlier code that computed the length of a string?

```
case List.head ["Dawn"] of
  Just name ->
    String.length name
  Nothing ->
    0
```

I think it looks nicer as a pipeline:

```
> ["Dawn"] |> List.head |> Maybe.map String.length |> Maybe.withDefault 0
4 : Int

> [      ] |> List.head |> Maybe.map String.length |> Maybe.withDefault 0
0 : Int
```

As you'll see in the chapter on The Elm Architecture, Elm browser apps typically have a type called `Model` to contain a representation of the app's state. A change (like a button click) invokes an `update` function that might change the model. Think of that as sending the model into a pipeline. Perhaps some function in the pipeline can produce a `Maybe`, which means that a `Nothing` must be handled. In that case, it makes sense for the default value to be the original model. That way, nothing changes if something went wrong. Like this:

```
model
  |> addSomethingTo             -- "changes" the model
  |> changeSomethingIn          -- "changes" the model
  |> trySomethingThatMightFail -- produces a `Maybe Model`
  |> Maybe.map continueWorking -- "changes" the wrapped model
  |> Maybe.withDefault model   -- extract the wrapped model, or no change.
```

To understand how this works, remember from earlier that all Elm objects are immutable. Functions that "change" the model create a new model. Here's some commentary on the "happy path" that might clarify:

```
model
  |> addSomethingTo             -- from `model`, we get a new `model1`
  |> changeSomethingIn          -- from `model1`, we get `model2`
  |> trySomethingThatMightFail -- from `model2`, we get a `Just model3`
  |> Maybe.map continueWorking -- from `Just model3`, we get `Just model4`
  |> Maybe.withDefault model   -- `Just model4` is unwrapped
```

And here's a sad path:

```
1 model
2   |> addSomethingTo             -- from `model`, we get a new `model1`
3   |> changeSomethingIn          -- from `model1`, we get `model2`
4   |> trySomethingThatMightFail -- the failure produces `Nothing`
5   |> Maybe.map continueWorking -- pass `Nothing` along
6   |> Maybe.withDefault model   -- because of `Nothing`, return original `model`
```

The name `model` on line 6 refers to the exact same structure as on line 1, which was not – and cannot be – changed by the pipeline.

This is pretty sweet, although a programmer focused on user experience might wonder whether a `Nothing` represents a problem the user (or someone) should know about. We'll cover such error-handling in a later chapter.

Haven't I seen `map` somewhere before?

Why yes, yes you have:

```
> List.map String.length [ "nineteen characters", "four" ]
[19,4] : List Int
```

Let's compare types:

```
> List.map
<function> : (a -> b) -> List  a  -> List  b
> Maybe.map
<function> : (a -> b) -> Maybe a  -> Maybe b
```

They look very similar, except that one calls for a "wrapper" that's a `List` and the other calls for a `Maybe`. This cries out for some abstraction, a way to say something like this:

1. There's an idea we'll call `Wrapperish`. Any type that wants to be considered `Wrapperish` has to have an associated `map` function with this type:

```
map : (a -> b) -> wrapperishThing a -> wrapperishThing b
```

Unfortunately, English only has two cases, upper and lower, or else I'd put `wrapperishThing` in a third. It's like a type variable in that it's a blank to be filled in with something more specific. It's different in that the "something" isn't exactly a concrete type. Instead it's something like `List` or `Maybe`. (We'll become more precise about this later.)

2. Both `List` and `Maybe` are `Wrapperish` because they have a `map` function that matches the abstract type above. So it seems we should be able to leave the `List` or `Maybe` off, use `map` all by itself:

```
map String.length ["1", "four"]  -- or ...
map String.length (Just "four")
```

... and have the compiler figure out we mean `List.map` in the first case and `Maybe.map` in the second.

Now, as it turns out, Elm doesn't allow this kind of generalization. (You have to write `Maybe.map`, not `map`.) So we'll cover this kind of polymorphism when we get to Purescript.

However, just because *Elm* doesn't know about it doesn't mean *we* can't think about it when creating our own types. If you're doing something "wrapperish," think about whether a `map` function of this sort would be useful. Above all, don't invent your own name when `map` is the name everyone already uses.

`map` functions should follow two guidelines. The guidelines are so natural that it's hard to construct functions that *don't* follow them, but they might be worth mentioning.

1. It would be weird if `map` worked on the wrapper rather than on the value(s) wrapped. That is, `map f (Just 3)` should be able to change the value `3` to produce something like `(Just "three")`, but it should *not* be able to produce `Nothing`.[2] And mapping a function over a list should not produce a list with a different size. A value should be mapped onto a value, leaving the wrapper out of it.
2. And we should require `map` to act sanely around function composition. Consider this:

```
> ["1", "bar"] |> List.map String.reverse |> List.map String.toUpper
["1","RAB"]
```

What if putting both the string functions within a single `map` behaved differently?

```
> ["1", "bar"] |> List.map (String.reverse >> String.toUpper)
["HI", "MOM!!!!!!!"]
```

We couldn't say we live in a sensible universe.

[2] I got this example from "AndrewC" of Stack Overflow at http://stackoverflow.com/a/13137359.

In the FP jargon, sanity-preserving rules like these are usually called *laws*. (The above two laws rejoice in the name "the functor laws," but forget that until we get to Purescript.)

I should note that Elm sometimes uses `map` for functions with slightly different types. Consider `String.map`:

```
> String.map Char.toUpper "aBAb"
"ABAB" : String
```

That looks like the familiar `map` behavior, and `String.map` obeys both laws, but look at its type compared to `List.map` and `Maybe.map`:

```
> String.map
<function> : (Char -> Char) -> String   -> String
> List.map
<function> : (a    ->    b) -> List a   -> List b
> Maybe.map
<function> : (a    ->    b) -> Maybe a  -> Maybe b
```

That doesn't match the required pattern because there aren't type variables. You can't convert a "string of characters" into a "string of numbers." Nevertheless, it's close enough that people familiar with `map`'s ubiquity will probably assume `String.map` exists without checking, and without worrying about the minor difference.

Lists of `Maybe`s

Suppose we have a list of strings. We want the first character of each string, ignoring strings that don't have first characters. That is, given this:

```
["a", "bad", "", "moon"]
```

... we want this:

```
['a', 'b', 'm']
```

(You know the values are characters because they're enclosed in single quotes.)

It's easy enough to split the strings:

```
> split = List.map String.uncons ["a", "bad", "", "moon"]
[Just ('a',""), Just ('b',"ad"), Nothing, Just ('m',"oon")]
```

And it's easy to fetch the characters of the non-`Nothing` results:

```
> almost = List.map (Maybe.map Tuple.first) split
[Just 'a', Just 'b', Nothing, Just 'm']
```

But now we have to somehow "squeeze out" the `Nothing` values. The default Elm `Maybe` functions don't provide an easy way to do that, but the community library `Maybe.Extra` does. The following `import` in effect adds its extra functions to `Maybe`:

```
> import Maybe.Extra as Maybe
```

... and now we have a `Maybe.values` function that does just what we want:

```
> Maybe.values almost
['a','b','m'] : List Char
```

So we can write another nice example of pipelining, one where `Maybe` is used to skip irrelevant steps, ending with a final step that moves values from the `Maybe` universe to the universe of plain values:

```
["a", "bad", "", "moon"]
  |> List.map String.uncons
  |> List.map (Maybe.map Tuple.first)
  |> Maybe.values
```

No muss, no fuss, no `case`.[3]

Multiple `Maybe` arguments

So far, we've been using `Maybe.map` to apply a function to a single argument. But what about functions that take two arguments? For example, consider the function `String.append` (known as `(++)` in its operator variant). Is there a way we can use it when the two arguments are `Maybe` types? Like this:

[3]Many people have observed that inheritance is just a way of feeding the type of an object into a potentially giant case statement that's hidden from the programmer by the language runtime. A similar hiding of `case` is happening here, just inside functions like `Maybe.map`. So much of design is about ways to make explicit case analysis go away.

```
> ??? String.append (Just "hi ") (Just "Dawn")
Just "hi Dawn" : Maybe.Maybe String

> ??? String.append (Just "hi ") Nothing
Nothing : Maybe.Maybe String
```

`Maybe.map` doesn't work. This:

```
> Maybe.map String.append (Just "hi ") (Just "Dawn")
```

… produces this:

```
-- TYPE MISMATCH ---------------------------------------

Function `map` is expecting 2 arguments, but was given 3.

5|   Maybe.map String.append (Just "hi ") (Just "Dawn")
                                          ^^^^^^^^^^^
```

There's actually a very simple answer. There are four additional variants of `map`: `map2`, `map3`, `map4`, and `map5`.[4] You just use the one appropriate to the number of arguments:

```
> Maybe.map2 String.append (Just "hi ") (Just "Dawn")
Just "hi Dawn" : Maybe.Maybe String

> Maybe.map2 String.append (Just "hi ") Nothing
Nothing : Maybe.Maybe String
```

If you have more than five arguments, you can implement your own `mapN`, but you might also want to consider if your function isn't doing too many things.

The `map` variants are a simple solution, but they're not exactly elegant. We'll see a different solution when we get to Purescript.

What now?

Let's look at sum types in a little more detail. I'll be more explicit about how they're defined and work, but mostly I want to concentrate on a few useful predefined ones and idiomatic uses.

However, before that, a now-scrapped exercise in this chapter made me realize that `Maybe.values` works as a way to introduce an important topic: designing with types.

[4]Optional and "rest" parameters (ones that gather extra arguments into a list) don't fit with this static FP way of thinking about functions. It's too bad: I find them really useful in languages like Elixir, Clojure, and Ruby.

6. A teaser: designing with types

Static FP people often say things like, “once you’ve got the types right, the code is easy.” What might that mean?

Consider the function `Maybe.Extra.values` from the previous chapter. It removes `Nothing` values from a `List` of `(Maybe a)` values and unwraps the remainder into a `(List a)`. This is its type signature:

```
values : List (Maybe a) -> List a
```

Suppose we were charged with implementing `values`. Because I’m comfortable with recursion and the `foldr` function, I could easily produce an implementation like the following. (You don’t need to understand it yet.)

```
values : List (Maybe a) -> List a
values maybes =
  let
    helper maybe accumulator =
      case maybe of
        Nothing -> accumulator
        Just maybe -> maybe :: accumulator
  in
    List.foldr helper [] maybes
```

That seems a bit complicated, so we could instead look around for some existing function that works with `Maybe` and lists. Perhaps such a function would do most of what we want, allowing a simpler implementation of `values`.

Here’s a candidate, `List.filterMap`:

```
filterMap : (input -> Maybe output) -> List input -> List output
```

It takes a function that produces `Maybe` values and applies it to an input list. `Just` values are unwrapped and included in the output. `Nothing` values are discarded. Here’s an example:

```
> List.filterMap List.head [ [], [1], [10, 20] ]
[1,10] : List number
```

Notice that the `Nothing` resulting from `(List.head [])` has been discarded.

Let's work through how thinking about types could help us use `filterMap` to implement `values`.

Both of these functions transform lists into lists. Let's line them up and see how they compare:

```
filterMap : (input -> Maybe output) -> List input     -> List output
values    :                            List (Maybe a) -> List a
```

Take a look at `filterMap`'s second parameter. Drop your eye down to the first `values` parameter. `input` can be anything. What if we specifically make it a `Maybe`? That's done by changing (or narrowing) both of `filterMap`'s uses of `input` to `(Maybe a)`:

```
filterMapUse : (Maybe a -> Maybe output) -> List (Maybe a) -> List output
values       :                              List (Maybe a) -> List a
```

(I've changed the name to `filterMapUse` because we're doing this to get ideas for how `filterMap` might be used to define `values`. We're not changing the types of the actual `filterMap`.)

`output` and `(a)` both mean "a value of any type," so we can do another substitution to make the two lines look more alike. To keep the line from wrapping, I'll choose to replace `output`:

```
filterMapUse : (Maybe a -> Maybe a) -> List (Maybe a) -> List a
values       :                         List (Maybe a) -> List a
                                       ^^^^^^^^^ hmm ^^^^^^^^^^
```

Those parallel `List ...` items sure look suspicious. Because the types are the same, it's *possible* that the corresponding values could be too. That is, the single argument to `values` might be passed as the second argument to `filterMap`, and the result of `filterMap` might just be the result of `values`. Like this:

```
values maybes = List.filterMap ??? maybes
```

To see if that actually works, we have to fill in the "hole" that I've represented with `???`. What function should go there? (Preferably one that already exists, but possibly one we have to write.)

The types can actually push us further toward deciding what the `???` function should do, but let's leave them aside for now. Here's an example that compares the `values` input to its output:

```
[ (Just 1) Nothing (Just 2) ]    -- input to `values`
[       1                 2  ]    -- output from `values`
```

1. If `(Just 1)` is fed to `???`, what should come out? `(Just 1)`, because it's `filterMap`'s job to unwrap the wrapped value and produce '1.
2. If `Nothing` is fed to `???`, what should come out? `Nothing`, because it's `filterMap`'s job to discard `Nothing` values.

So in both cases, we want `???` to do nothing. There's a function that does that: `identity`. You implemented it, in fact.

We're done. Here's the definition of `values`, complete with type annotation:

```
values : List (Maybe a) -> List a
values maybes = List.filterMap identity maybes
```

This is a case where I'd be pretty comfortable with point-free style:

```
values : List (Maybe a) -> List a
values = List.filterMap identity
```

… mainly because the type annotation provides all the information that the name `maybes` suggests.

Taking "type-driven" all the way

Let's get to the definition of `values` a different way: using types. Let's return to the implementation with the hole:

```
values : List (Maybe a) -> List a
values maybes = List.filterMap ??? maybes
```

We know the type of `???` is `(Maybe a -> Maybe a)`. Let's consider the possibilities.

If the argument is `Nothing`, the result can either be `Nothing` or some `(Just a)` value. Suppose we want it to be a `(Just a)` value. How do we calculate it? What value do we wrap with the `Just`?

There's actually nothing we can do. When `values` is applied, it's applied to a real value, with some concrete type. But at compile time, we can't predict what that concrete type will be. It could be anything. Therefore, any value we use in place of `Nothing` will sometimes be wrong. For example, we couldn't decide to always return `(Just 5.3)`. If we tried, Elm would complain:

```
-- TYPE MISMATCH ------------------------------------------

The definition of `values` does not match its type annotation.

37| values : List (Maybe a) -> List a
38|>values maybes = List.filterMap (always (Just 5.3)) maybes

The type annotation for `values` says it always returns:

    List a

But the returned value (shown above) is a:

    List Float
```

The type annotation says that if you pass in a `List (Maybe String)`, you're supposed to get a `List String` out. But the `values` implementation converts a `Nothing` to a `Float`. So it must always convert a list – no matter the element type – to a `List Float`. That's not what the type annotation promises.

Therefore, there's no choice but for `???` to convert a `Nothing` to a `Nothing`.

What about the case where `???` is given a `Just a`? The same problem arises. Without knowing the actual type of the argument, `???` has nothing it can do. It can't replace the wrapped value with a different wrapped value. There's no function in Elm that takes a value of an arbitrary type and returns a different value of that type. (For example, `negate` only works on numbers.)

`???` can't do anything else to the wrapped value either. For example, it can't wrap it in a list because that would change the type of its output from `(Maybe a)` to `(Maybe (List a))`.

We're stuck with an identity function.

What now?

That looks like a contrived example, but it's mostly true. I did write an exercise that instructed you to implement `values`. I did realize that I hadn't taught you recursion or "folding." I did search through the `Maybe`, `Maybe.Extra`, and `List` APIs to look for likely helper functions. I did seize on `List.filterMap` based on its type signature. But after I found it, I didn't proceed so methodically, and I doubt my train of thought followed the steps here. (I thought these steps showed off the design moves best.) I pretty much jumped directly to `identity`.

I want to highlight two things:

1. You can use types as a way to find helper functions.
2. Types can give you the rough “shape” of a function. That gies you the option of breaking coding into two steps: letting types drive some choices, then filling in the remaining blanks.

I hope this example whets your appetite. I’ll return to this theme at various points throughout the book.

In the meantime, it’s time to end this digression and move on to “sum types,” as promised in the `Maybe` chapter.

7. Roughly 23 more chapters on Elm go here

See leanpub.com for the full table of contents.

PureScript chapters begin after the Elm chapters. I’m including Chapter 30 in this free sample in the hopes that it might be useful for Elm programmers thinking of making the leap. It describes PureScript features or idioms that are essentially the same as Elm’s, but have differences that might trip you up as you learn PureScript.

Anyone who tries to use the PureScript repl or write a runnable PureScript program will run across *type classes*, and so I describe them in chapter 30. Also for that reason, I’m including versions of chapters 25, 26, and 27. 25 and 27 introduce type classes using Elm terminology. 26 explains JSON parsing in a way that’s used in 27. Although I’ve edited these, they still contain references to material this sample doesn’t include. You may have to do some skimming or learn an Elm topic you don’t already know.

8. Monoids, laws, and the avoidance of creepiness

Up until now, type checking has been about matching functions to concrete data. Every piece of data labeled `String` is guaranteed to have a certain set of functions (`toUpper`, `map`, `toInt`) that work with it. In this chapter, we'll build on the earlier observation that `map` seems to do the same(ish) kind of thing for vastly different types. I'll try to convert the vague "same(ish)" into something more specific, while still avoiding precise definitions and technical details.

This material will mostly only be of conceptual use right now, as Elm doesn't have the mechanism to implement the ideas. I'll show those later, in PureScript. However, it does introduce a way of thinking about design that can be of use even in Elm.

Lens composition behaves sensibly

Material here is omitted

Motivating monoids

`List`, `String`, and `Array` all have an `append` function. In all three cases, it's associative:[1]

```
> String.append "a" (String.append "b"  "c")
"abc" : String

> String.append (String.append "a" "b") "c"
"abc" : String
```

`List` and `String` also have a `concat` function for appending a whole list of values:

```
> String.concat [ "a", "b", "c" ]
"abc" : String
```

But `Array` doesn't. Hmm. I guess no one wrote it. That's peculiar, but it's easy to fix with `foldl`. It could be put in the same module as all the other array functions:

[1]In the case of `List` and `String`, the operator `++` is a synonym for `append`. `Array` doesn't have that synonym, so I'll stick to the function.

```
module Array ...

concat : List (Array a) -> Array a
concat list =
  {- Because `foldl` puts the accumulator second,
     we use `flip` so that the first `foldl` step
     for this:
             concat [ [1, 2], [3], [4, 5] ]
     ... calls `append` like this:
             append [] [1, 2]
  -}
  List.foldl (flip append) empty list
```

Introducing `foldr`

The need to `flip` the arguments to `append` is annoying, but it's also easily avoided by using the alternate fold `foldr`, which starts with the *last* (rightmost) element in the list rather than the first (as `foldl` does). The `foldr` version looks like this:

```
concat : List (Array a) -> Array a
concat list =
  List.foldr append empty list
```

To see how `foldr` works, consider `concat [ [1, 2], [3], [4, 5] ]`. (I'm writing the arrays in list notation for readability.) This would be the sequence of calls to `append`:

```
            append empty [4, 5]
        append [3]       [4, 5]
 append [1, 2]        [3, 4, 5]
                [1, 2, 3, 4, 5]
```

Note: because `foldl` accumulates from the left and `foldr` from the right, the two implementations are equivalent only because `append` is associative.

`Array.empty` is used as the starting value of the accumulator. For that to work, it has to work with `append` in a specific way:

```
> append empty     [1, 2, 3]  -- left identity
[1,2,3]
> append [1, 2, 3] empty      -- right identity
[1,2,3]
```

(If you want to be picky, our `Array.concat` implementation requires only the first, but having that but not the second would be creepy.)

Left and right identity are laws like those of the previous chapter. A value that obeys them is called an *identity* or a *unit*.

So:

- Using `(List.foldr append empty)` to implement `Array.concat` produces sensible results *because* `Array.append` is associative *and* `Array.empty` is an identity element.

But there's nothing special about `Array`. We could replace `Array` in the sentence above with other types:

- Using `(List.foldr append empty)` to implement `String.concat` produces sensible results *because* `String.append` is associative *and* `String.empty` is an identity element.
- Using `(List.foldr append empty)` to implement `List.concat` produces sensible results *because* `List.append` is associative *and* `List.empty` is an identity element.

At least, we could if the identity elements for `List` and `String` were spelled `empty` instead of `[]` and `""`. But it would be easy to add those synonyms, allowing us to generalize three claims about specific types into one:

- Using `(List.foldr append empty)` to implement *Type*`.concat` produces sensible results whenever *Type*`.append` is associative and *Type*`.empty` is an identity element.

The phrase "is associative … and … identity element" is cumbersome, so we'll use the mathematician's shorthand for it: *monoid*. So:

- Using `(List.foldr append empty)` to implement *Type*`.concat` produces sensible results when *Type* is a monoid.

Declaring `Monoid`

Let's summarize what we know about monoids in a codelike form. Here's the first of four parts:

```
module Monoid exposing (..)

type class Monoid =
  ...
```

`Monoid` is conventionally written capitalized, like types are, but it's not a type. It's instead a partial description of any number of types. Such partial descriptions are typically called *type classes*.

The word "class" in "type class" has nothing to do with classes in object-oriented programming. I think it's just a coincidence. English doesn't have enough words that mean "things that are alike in some important way," so it's inevitable that different people will use words like "class" and "kind" to mean different things. (It's fortunate that the word "type" got snatched up early enough in the history of computing that everyone uses it in a pretty much compatible way.)

The next part describes components that all monoids require:

```
requires
  append : ty -> ty -> ty  | ty of Monoid
  empty :  ty              | ty of Monoid
```

Here, `ty` is a type variable, just like always, but the latter part of the definition restricts it to being a `Monoid` type. Notice there's no code for `append`. That's because individual `Monoid` types (like `String`) will have different implementations for `append` and `empty`.

So far, there's nothing about the laws monoids must follow (identity and associativity). We could write those laws as comments but, as with the lens laws from the previous chapter, we can also provide the information in executable tests:

```
checkWith
  leftIdentity : ty -> Test             | ty of Monoid
  leftIdentity x =
    equal_ (append empty x) x

  rightIdentity : ty -> Test            | ty of Monoid
  rightIdentity x =
    equal_ (append x empty) x

  associative : ty -> ty -> ty -> Test  | ty of Monoid
  associative x y z =
    let
      left =  append (append x y) z
      right = append x (append y  z)
    in
      equal_ left right
```

So someone writing a `BitSequence` type could test sample values against the monoid laws:

```
laws =
  let
    bits = BitSequence.fromString
  in
    describe "monoid laws"
      [ Monoid.leftIdentity  <| bits "0101"
      , Monoid.rightIdentity <| bits "0111"
      , Monoid.associative
         (bits "1") (bits "10") (bits "001")
      ]
```

… and be more confident in labeling `BitSequence` a monoid.

Property tests

Notice that I tested the laws with five different bit strings. It's generally a good idea to vary your test values, just on the off chance that you stumble over some peculiar edge case.

Property tests mechanize this idea. If you have a way to generate many random strings of `'0'` and `'1'`, you can easily test `leftIdentity` with 100 different bit strings:

```
List.map Monoid.leftIdentity (bitStrings 100)
```

Property tests are an economic decision[a]. If someone has already provided you with a `bitStrings` generator, why not use it? It costs almost nothing, compared to hard-coding a single test. Moreover, perhaps the person who wrote the generator is a better tester than you and included special values you'd miss, like the empty string and a string so long that it wouldn't fit in an Elm integer.

On the other hand, if *you* have to write the generator, you should first consider its incremental value. How likely is it that 100 generated strings will find a bug that "0101" doesn't? Folklore and simulation evidence[b] suggest that – in many cases – it's not so much the random tests that provide value, but rather hardcoded edge cases like the empty string.

I haven't decided if the final version of this book should talk more about generative testing. It's certainly sometimes useful, and certainly very popular in the static FP world, but I think there's a certain faddishness to it, a certain amount of herd-following.

[a] In 1988, I presented a paper on the economics of testing called "When should a test be automated?". I think the structure of the analysis still works, even though the specifics of the example are vastly different. (Automated vs. manual tests then; property-based vs. ordinarily-automated tests now.)

[b] See Hamlet and Taylor, "Partition testing does not inspire confidence", IEEE Trans. Software. Eng. SE-16 (December, 1990) At the time of writing, I can't find a version you don't have to pay for. Grr.

The final bundle of `Monoid` information gives functions that work for any type that implements the required functions and follows the laws:

```
  provides
    concat : List ty -> ty                     | ty of Monoid
    concat list =
      List.foldr append empty list

    power : ty -> Int -> ty                    | ty of Monoid
    power x i = concat (List.repeat i x)

    foldMap : (a -> ty) -> List a -> ty        | ty of Monoid
    foldMap f =
      List.foldl (\a acc -> append acc (f a)) empty
```

Someone implementing `BitSequence` could copy the definitions of `concat`, `power`, and `foldMap` into the `BitSequence.elm` module, feeling confident that they'd behave sensibly. Being lazy, though, we programmers would prefer that the compiler did the work for us. We'll soon look at how to tell the compiler to do that, but first let's look at those last two functions. They can be part of any module that declares a `Monoid` type.[2]

Use `power` when all the values you want to `concat` are the same:

```
> List.power [1, 2] 4
[1,2,1,2,1,2,1,2]

> String.power "ab" 4
"abababab"
```

`foldMap` is a third fold function. It's simpler than the other two when you're accumulating transformed values into a monoid. That is, instead of writing a fold like this:

```
> List.foldl (\a acc -> List.append acc (List.repeat a a)) [] [1, 2, 3]
[1,2,2,3,3,3] : List Int
```

… you can write it like this:

```
> List.foldMap (\a -> List.repeat a a) [1, 2, 3]
[1,2,2,3,3,3] : List Int
```

Mapping a function over a list and accumulating the results into a monoid is common enough that it's worth having a function just for that.

Perhaps that's confusing. As another example, pretend that the `Sum` type looks like this:

[2] I'm simplifying here to make a better example. `foldMap` is actually better placed in a `Foldable` type class. All that means is that types that want it need to declare themselves `Foldable` in addition to `Monoid`. But let's deal with types that implement multiple type classes in a later part of the book.

```
type Sum = Sum Int
```

It can be defined as a `Monoid` with `identity` 0 and `append (+)`.[3] That given, we can use `foldMap` to sum up a list of lists:

```
> lists = [ [1], [3, 3, 3], [2, 2], [] ]

> Sum.foldMap List.length lists
6: Int
```

Once you've absorbed idioms of static FP, such code is easier to understand than a more detailed solution using `foldr` or `foldl`. (But not before!)

Using `Monoid`

For simplicity, let's implement a `BitSequence` as a tagged `String` and ignore invalid strings.

```
module BitSequence exposing (BitSequence, ...)

type BitSequence = Bits String
```

We want `BitSequence` to be a `Monoid`, which means we need to define `empty` and `append`:

```
type class instance BitSequence of Monoid where
  empty = Bits ""

  append (Bits one) (Bits two) =
    Bits <| one ++ two
```

Notice that we don't have to provide a type annotation for `empty` and `append` because those are provided in the `Monoid` definition.

Now the compiler can provide `concat`, `power`, and `foldMap`. And we can use them in the implementation of other `BitSequence` functions as well as expose them to client modules.

Here, for example, is client code that converts a `String` into a `BitSequence`, where `1` means the corresponding character is a digit:

[3]The `Sum` type is needed because integers support two monoids: one with `0` and `(+)`, one with `1` and `(*)`. `Sum` is used to explain which one we want.

```
String.toList "832h3"
  |> BitSequence.foldMap (Char.isDigit >> BitSequence.fromBool)
Bits "11101" : BitSequence
```

In the above, the function `fromBool` is another `BitSequence` constructor:

```
fromBool : Bool -> BitSequence
fromBool b =
  case b of
    True -> Bits "1"
    False -> Bits "0"
```

Monoid instances name types

When we say "arrays are monoids," what we mean is:

```
type class instance (Array a) of Monoid where
  empty = ...
  append = ...
```

Note that we use the quantified type `Array a` rather than `Array`. I'll say more about that when we discuss JSON decoders and functors.

Spot the `Monoid`

Part of the skill of programming in static FP languages is discovering that a new type of yours can fit into a type class that gives you functions for free.

For example, with appropriate definitions of `empty` and `append`, `Maybe` can also be a monoid. Let's decide to make the identity (`empty`) `Nothing`. Then consider this definition for `append`:

```
append : Maybe a -> Maybe a -> Maybe a
append left right =
  case left of
    Nothing -> right
    _ -> left
```

Here, "appending" two `Maybe` values produces the leftmost `Just` or `Nothing`:

```
> Maybe.append (Just 1) (Just 2)
Just 1 : Maybe number

> Maybe.append Nothing (Just 2)
Just 2 : Maybe number

> Maybe.append Nothing Nothing
Nothing : Maybe a
```

Given those definitions, `concat` would produce the leftmost `Just` of a list of `Maybe` values (or `Nothing`):

```
> concat [ Just 1, Nothing, Just 3 ]
Just 1 : Maybe number
```

Notice that `concat` produces the sensible result for the empty list: `Nothing`. One of the consequences of working with law-obeying types is that edge cases tend to behave sensibly.

But that's not the only possible monoid for `Maybe`. This variation:

```
append : Maybe a -> Maybe a -> Maybe a
append left right =
  case right of
    Nothing -> left
    _ -> right
```

… prefers the right `Just` over the left. So, `concat` will produce the rightmost `Just` in a list (or `Nothing`).

Naming

You may note that `Maybe.concat` is a pretty bad name. `Maybe.rightmost` would be better.

This situation is common with type classes. It's probably an inevitable consequence of the development of ideas like "monoid":

1. People notice that the colloquial idea of "concatenation" applies to a variety of types: strings, arrays, lists.
2. People abstract from the examples to discover the laws underlying the similarity.
3. People discover that the laws apply to many new types, including ones where the original name ("concatenation") applies poorly.

There are two ways to handle this awkwardness:

1. Stick with the original names. Assume that they won't make it harder for people to understand the underlying laws, or to extend the idea to new types. Suggest that it's not a horrible idea to add `rightmost` as a synonym for `Maybe.concat`.
2. Abandon the original names. Favor teaching abstractions in terms of laws, replacing words like "append" with arbitrary names or symbols. For example, you might drop `append` and only use an operator like `<>`. That way, preconceptions ("concatenation is about sequences of things") don't limit thinking.

In principle, I'm sympathetic to the second view, but it doesn't work for me. I learn more quickly the first way, and I'm willing to accept some shallowness of understanding in exchange. So this book will tend to explain by replaying the 1-2-3 sequence of discovery.

Now look at this third definition for a `Maybe.append`:

```
append left right =
  case (left, right) of
    (Nothing, Nothing) -> Nothing
    (Nothing, right) -> right
    (left, Nothing) -> left
    (Just a, Just b) -> Just (append a b)
```

What, now, will `Maybe.concat` do?

Answer on the next page.

It concatenates the non-`Nothing` values, if any, wrapping the result in a `Just`:

```
> concat [ Just "a", Nothing, Just "b", Nothing ]
Just "ab" : Maybe String

> concat [ Nothing, Nothing ]
Nothing : Maybe a
```

Function composition is not quite a `Monoid`

The remainder of the chapter is omitted.

What now?

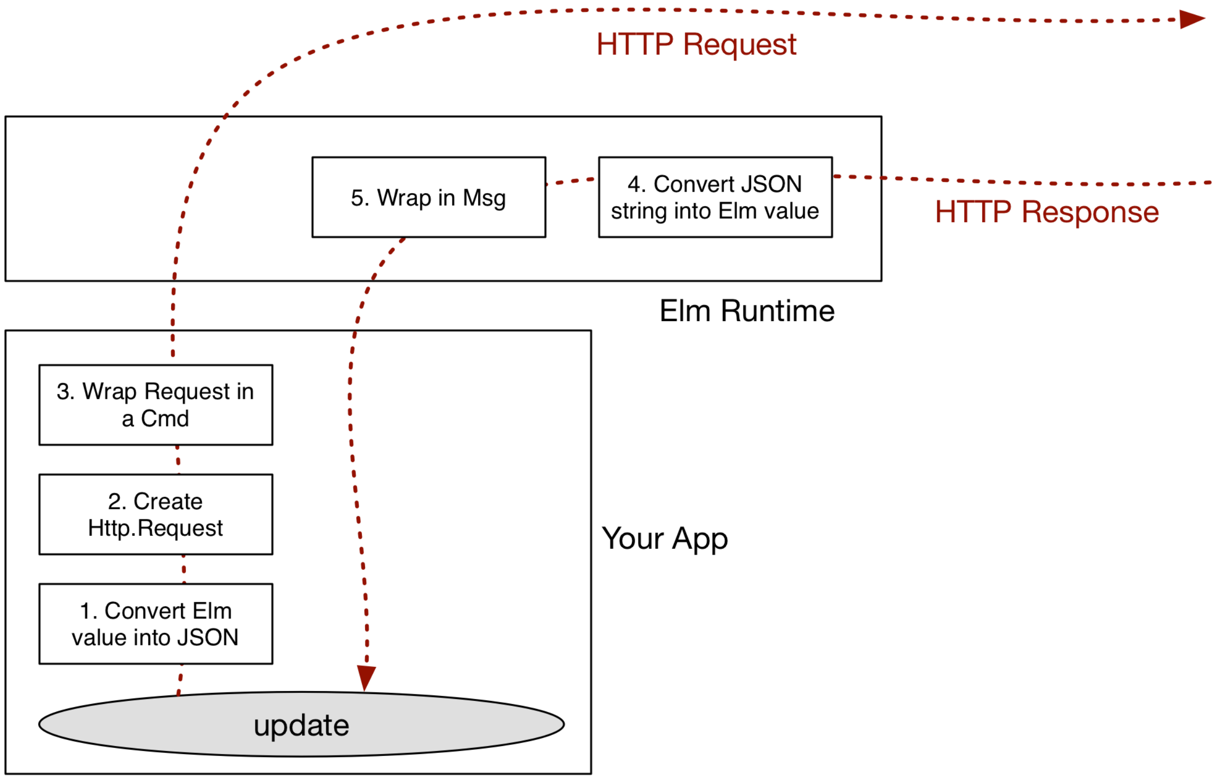
After a chapter like this one – about ideas much more than about actual code – it should be refreshing to look at how lenses can be used to simplify application error handling.

9. Error handling, pipelines, and JSON

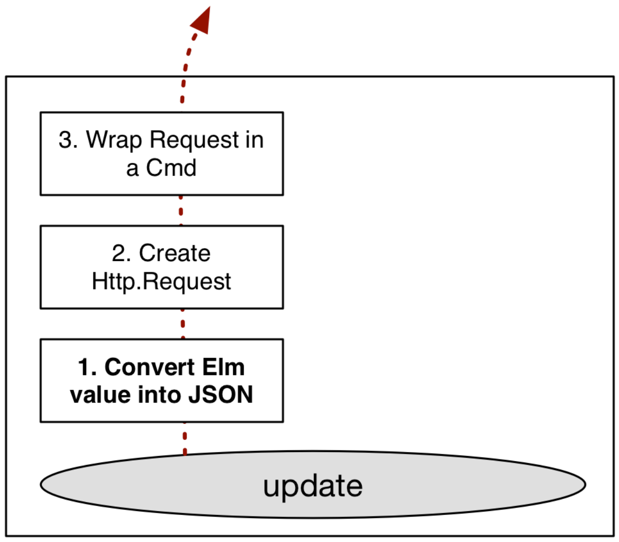
The bulk of this chapter is omitted. I'm retaining the part that explains JSON error handling, because it's used in the next chapter.

Reporting an error via HTTP

We want to HTTP POST a JSON version of an error to an external server. Such a side effect can't be done within a pure FP language, so the Elm runtime has to do it. Here's how:



1. Convert an Elm value into JSON



JSON is easier in dynamically-typed languages like Ruby or Clojure. Because such languages can ask a value what its type is, it's straightforward to write a function that can convert an arbitrary structure to JSON:

```
irb> {:a => [1, 2, 3]}.to_json
=> "{\"a\":[1,2,3]}"
```

Elm has no way of doing such "type introspection," so you must tell JSON-producing code what "shape" of data it will be given. Consider these Elm record values:

```
{a = [1, 2, 3]}
{a = []}
```

If we want to encode them, we have to write a specific function:

```
1  import Json.Encode as Json
2
3  recToValue : { a : List Int } -> Json.Value
4  recToValue rec =
5    let
6      listValues = List.map Json.int rec.a
7      listValue = Json.list listValues
8    in
9      Json.object [ ( "a", listValue ) ]
```

line 6:
All of the `Json` functions convert an Elm value to a `Json.Value` (an opaque type). This line shows the `(Json.int : Int -> Value)` function mapped over a `List Int` to produce a `List Json.Value`.

line 7:
A `List` of `Json.Values` is converted to a `Json.Value` with `Json.list`.

line 9:
The closest JavaScript analogy to an Elm record is an object (a set of key/value pairs). The usual list-of-pairs trick is used to allow heterogenous field types.

line 3:
The function's return type is `Json.Value` rather than a JSON string. That way, it can be combined with other functions to encode even larger JSON structures.

The final string is produced with `Json.encode`:

```
> recToValue rec |> Json.encode 0
"{\"a\":[1,2,3]}" : String
```

(The `0` argument to `encode` produces the shortest possible JSON string. Larger numbers produce a multi-line string with indentation.)

Here's another example of JSON encoding, from Errors.Remote.Errors:

```
1 pathErr : String -> State Error -> BadPath -> Json.Value
2 pathErr errorId state path =
3   Json.object
4     [ ("app", Json.string "Errors.Remote.Errors")
5     , ("id", Json.string errorId) -- "MissingWord", for example
6     , ("msg", Json.string <| toString state.msg)
7     , ("path", Json.list <| List.map Json.string path)
8     ]
```

Notice that, on line 6, the full `State` is used to extract relevant context (the `Msg` that provoked the error).

More text omitted here

Interlude: some basics of JSON decoding

The HTTP Response contains string data and that data is usually in JSON format. So it needs to be decoded. In the general case, an Elm decoder is built from a DSL that mirrors the structure of the JSON (in a way reminiscent of how views use a DSL that mimics HTML).

Elm's approach to JSON is not representative of all static FP languages. Other languages (with more expressive/complicated type systems) require less work on your part. We'll see examples of that later.

Decoding works with composable values of type `(Json.Decode.Decoder String)`. Here's how you decode a float:

```
> Decode.decodeString Decode.float "5.3"
Ok 5.3 : Result String Float
```

Here's how you *fail* to decode a float:

```
> Decode.decodeString Decode.float ""
Err "Given an invalid JSON: Unexpected end of JSON input"
    : Result String Float
```

Here's a decoder for a single field in a record:

```
> decodeXFloat = Decode.field "x" Decode.float
<decoder> : Decoder Float

> Decode.decodeString decodeXFloat "{\"x\":3.3}"
Ok 3.3 : Result String Float
```

Notice that the result isn't a record but rather the field value. To make a record, we have to apply a function to that value. To show how that works, I'll use an Elm-specific feature that I've avoided until now. A type alias for a record creates a constructor of the same name as the type. Given this:

```
type alias X = {x : Float}
```

… you can create an `X` like this:

```
> X 3.3
{ x = 3.3 } : X
```

So the equivalent JSON structure could be decoded like this:

```
> json = "{\"x\":3.3}"
> json                                  \
    |> Decode.decodeString decodeXFloat  \
    |> Result.map X
Ok { x = 3.3 } : Result String X
```

While that works, it's awkward:

1. The structure of the code doesn't look much like the structure of the data.
2. There's no obvious way to write a `decodeX` function that can be composed with other decoders.

Instead, `Json.Decode` has its own `map` function that converts one decoder into another:

```
> decodeX = Decode.map X decodeXFloat
<decoder> : Decoder X

> Decode.decodeString decodeX json
Ok { x = 3.3 } : Result String X
```

Just as there are `map2` functions for other types, there's one for `Json.Decode`. It can be used to build larger product types. Consider a point that's represented (in JSON) by a two-element array:

```
> json = "[1.3,2.8]"
```

`Decode.index` is used to pull individual elements from a JSON array. Here's how you get the 0th element:

```
> get0 = Decode.index 0 Decode.float
<decoder> : Decoder Float
> Decode.decodeString get0 json
Ok 1.3 : Result String Float
```

The constructor for a two-element tuple is the operator `(,)`:

```
> (,) 1 2
(1,2) : ( number, number1 )
```

We can use that operator to build a point `Decoder` by mapping it over two `Decode.index` decoders:

```
> decodePoint =                         \
    Decode.map2 (,)                     \
      (Decode.index 0 Decode.float)     \
      (Decode.index 1 Decode.float)
<decoder> : Decoder ( Float, Float )

> Decode.decodeString decodePoint json
Ok (1.3,2.8) : Result String ( Float, Float )
```

Elm decoders in practice

It can be tedious and error-prone to write such composed decoders. Rather than repeat what's been written elsewhere, let me point to places that can help you with that task:

"How JSON decoding works in Elm," by Kofi Gumbs. Part 1 – Part 2 – Part 3,

"Demystifying Elm JSON decoders," by Ilias Van Peer,

"Decoding JSON structures with Elm," by Josh Clayton,

A cheatsheet by Lim Yang Wei,

… and an ebook, *The JSON Survival Kit*, by Brian Hicks.

More omissions

What now?

Let's go abstract and think about a type class that describes something `String`, `Array`, and `Json.Decode.Decoder` have in common: `map`.

10. JSON decoders, functors, and type constructors

In the previous chapter you saw how to convert a JSON number into a record wrapping an `Float` by mapping the record's constructor (line 4):

```
1 > decodeXfloat = Decode.field "x" Decode.float
2 <decoder> : Decoder Float
3
4 > Decode.map X decodeXFloat
5 <decoder> : Decoder X
```

`Decode.map` looks like `Maybe.map`:

```
Decode.map : (a -> b) -> Decoder a -> Decoder b
Maybe.map  : (a -> b) -> Maybe   a -> Maybe   b
```

So that's good.

Other versions of `map` can be used to change a value:

```
> Maybe.map negate (Just 3)
Just -3 : Maybe number

> List.map negate [1, 3]
[-1,-3] : List number
```

... or to change a type:

```
> Maybe.map toString (Just 3)
Just "3" : Maybe String

> List.map toString [1, 3]
["1","3"] : List String
```

`Decode.map` has almost the same behavior, but not quite. A separate step is needed to see the effect of the transformation. In the following, there's no evidence that `(Decode.map negate)` does anything at all until lines 7 and 8:

```
1 > Decode.int
2 <decoder> : Decoder Int
3
4 > decoder = Decode.map negate Decode.int
5 <decoder> : Decoder Int
6
7 > Decode.decodeString decoder "3"
8 Ok -3 : Result String Int
```

In the case where `Decode.map` changes the type, we see evidence of change sooner (line 2 shows a `String` not an `Int`), but it still seems the transformation is untested until line 4:

```
1 > decoder = Decode.map toString Decode.int
2 <decoder> : Decoder String
3
4 > Decode.decodeString  "3"
5 Ok "3" : Result.Result String String
```

Does this different behavior mean a `Decoder` is somehow fundamentally a different kind of thing than a `List`? Or could it fit together with them into the same type class, one called *functor*?[1]

> Note that I'm *not* asking whether a `Decoder` is a monoid. It's definitely not: it has neither `empty` nor `append`. But a type can be an instance of many type classes, just like a Java class can implement many interfaces or a Ruby class can include many modules. `List` and `Maybe` are both monoids and functors.

In order to see whether `Decoder` is a functor, despite its peculiarity, we'll first have to define `Functor`, work through a pair of functor types that are a bit more surprising than `List` and `Maybe`, and finish up by (spoiler!) deciding that a `Decoder` is indeed a functor.

`Functor` and its laws

`map` is the single required function for the `Functor` type class. Its definition could start like this:

[1]It's possible that the word "functor" in mathematics comes from the word "functor" in linguistics. There, functors are "the glue that holds sentences together" (pronouns, words like "the", "however", "okay", and "uh"). The connection between that and the role of functors in programming is… unclear.

```
type class Functor =
  requires
    map : (a -> b) -> lifted a -> lifted b     | lifted of Functor
```

I'm using "lifted" to continue the metaphor of values being moved from one universe to a more complex one. (In the case of `Maybe`, `Just` lifts a value into a universe with one additional `Nothing` value. A `Decoder` lifts a type into the universe of decoders for types.)

Here's the `map` function for the two familiar instances of `Functor`:

```
      map : (a -> b) -> lifted a  -> lifted b  | lifted of Functor
  List.map : (a -> b) -> List   a  -> List   b
 Maybe.map : (a -> b) -> Maybe  a  -> Maybe  b
```

And here are the functor laws that those instances obey, written as tests:

```
  checkWith
    -- mapping `identity` does nothing.
    identity: lifted a -> Test        | lifted of Functor
    identity x =
      equal_ (map identity x) x

    -- mapping a composition of two functions is the same
    -- as composing two mappings.
    composition : (a -> b) -> (b -> c) -> lifted a -> Test
                                      | lifted of Functor
    composition f g x =
      let
        left =  map (f >>     g)
        right = map   f >> map g
      in
        equal_  (left x) (right x)
```

Here are examples of why `List` and `Maybe` are functors:

```
> Maybe.map identity Nothing == Nothing
True : Bool

> left = List.map (negate >> round)
> right = List.map negate >> List.map round
> left [1.1, 8.8] == right [1.1, 8.8]
True : Bool
```

What's up with `Result`? (type constructors)

`Result` also seems to be a functor. It has the right "shape" of `map`:

```
       map : (a -> b) -> lifted     a  -> lifted     b  | lifted of Functor
Result.map : (a -> b) -> Result err a  -> Result err b
```

And it obeys the laws:

```
> Result.map identity (Ok 3) == Ok 3
True : Bool
> Result.map identity (Err "!") == Err "!"
True : Bool
> Result.map negate (Ok 3) == Ok -3
True : Bool
> Result.map negate (Err "!") == Err "!"
True : Bool
```

The last two expressions reveal an oddity. Why does `map` only apply to values wrapped in `Ok`? Why not `Err` instead? Or why not `Err` as well (mapping both values)?

Earlier in the book, I answered the way I would my young daughter Sophie after her 50th "but why…?" in ten minutes: Because I said so. That's just the way it is. Now I'll show you why that's the way it *has to be*.

Type constructors

Let's compare required functions for `Monoid` and `Functor`:

```
1 type class Monoid =
2   requires
3     append : ty -> ty -> ty  | ty of Monoid
4
5 type class Functor =
6   requires
7     map : (a -> b) -> lifted a -> lifted b  | lifted of Functor
```

On line 3, `ty` is something that can be replaced by `String` or `List a` or `List Int`. That is: it refers to a type.

But line 7's `lifted` is replaced by `List` or `Array` or `Decoder`. That is: it refers to a *type constructor*. Type constructors are function-like things that convert one type to another. `List` can convert `Int` to `List Int`.

Compare type constructors to value constructors:

- The value constructor `Just` converts `3` to `Just 3`.
- The type constructor `Maybe` converts `Int` to `Maybe Int`

And:

- `Just` applies at runtime.
- `Maybe` applies at compile time.

When we create an instance of a `Monoid` type class, we provide a type:

```
type class instance (Array a) of Monoid where ...
```

... because we want this substitution for `ty`:

```
append : ty      -> ty      -> ty
append : Array a -> Array a -> Array a
```

But when we want to create an instance of `Functor`, we provide a type constructor:

```
type class instance Array of Functor where ...
```

... because we want this substitution for `lifted`:

```
map  : (a -> b) -> lifted a -> lifted b
map  : (a -> b) ->  Array a ->  Array b
```

With that background out of the way, let's consider the `Result` type constructor. It seems to take two arguments:

```
type Result err ok = ...
```

But, like all functions, that's an illusion. The illusion can be broken with parentheses:

```
type (Result err) ok = ...
```

... which means that `(Result String)` produces a type constructor that takes a single type – `Int`, let's say – and produces a new type `(Result String Int)`.

Therefore, if we want to create an instance of `Functor` for `Result`, we need to provide the `(Result err)` type constructor. We can't give it `Result` alone, because that's a constructor of type constructors. And there's no equivalent of `flip` that lets us rearrange `Result`'s parameters. So we're stuck with this:

```
type class instance (Result err) of Functor where ...
```

Substituting `(Result err)` for `lifted` gives us this:

```
map  : (a -> b) ->       lifted a ->       lifted b
map  : (a -> b) -> (Result err) a -> (Result err) b
-- or
map  : (a -> b) ->   Result err a ->   Result err b
```

Exercise

Consider this type:

```
data Silly a b
  = AsA a
  | AsB b
  | AsAB a b
  | Serious
```

Can that type be a functor? If so, predict the results of the following:

```
> map negate (AsA 3)
> map negate (AsB 3)
> map negate (AsAB 3 333)
> map negate Serious
```

My opinion is after the page break.

Yes, it can. This would be the instance declaration:

```
type class instance (Silly a) of Functor where ...
```

And these would be the results:

```
> map negate (AsA 3) -- doesn't use `b` variable
AsA 3
> map negate (AsB 3) -- does use it
AsB -3
> map negate (AsAB 3 333)
AsB 3,-333
> map negate Serious
Serious
```

An Elmish implementation of `Silly`'s `map` would look like this:

```
map : (a -> b) -> Silly a -> Silly b
map f silly =
  case silly of
    AsB b ->
      AsB (f b)
    AsAb a b ->
      AsAb a (f b)
    noB ->        -- default case
      noB
```

You can make a type with any number of type variables a functor, but if the `map` is to be valid, it must always apply to the last type.

Are functions functors?

Omitted. (The answer is Yes.)

Are JSON decoders functors?

Does `Decode.map` match the type annotation?

```
       map : (a -> b) ->  lifted a ->  lifted b
Decode.map : (a -> b) -> Decoder a -> Decoder b
```

Yes.

Does `Decoder` obey the laws?

I have pretty strong confidence it obeys identity:

```
> original = Decode.float
> mapped = Decode.map identity Decode.float

> Decode.decodeString original "38.0000000001"
Ok 38.0000000001 : Result String Float
> Decode.decodeString mapped   "38.0000000001"
Ok 38.0000000001 : Result String Float

> Decode.decodeString original "hi dawn"
Err "Given an invalid JSON: Unexpected token h in JSON at position 0"
    : Result String Float
> Decode.decodeString mapped "hi dawn"
Err "Given an invalid JSON: Unexpected token h in JSON at position 0"
    : Result String Float
```

You can try as many examples as you need, with various different decoders, to gain your own confidence. Or you can look at the source to convince yourself.

What about the composition of two functions, `f` and `g`?

```
> f = (*) 5
> g = (+) 1

> one = Decode.int |>  Decode.map (f >>            g)
> two = Decode.int |> (Decode.map  f >> Decode.map g)

> Decode.decodeString one "4"
Ok 21 : Result String Int
> Decode.decodeString two "4"
Ok 21 : Result String Int
```

That's convincing enough for me, especially since I suspect that it would be harder to make a broken implementation than a working one.

(If you need more convincing, perhaps the property tests I wrote for `Decoder` will do the trick: tests/TypeClass/DecodeTest.elm. Comments there explain how the tests work.)

Using functors

In languages with type classes, many many types are instances of `Functor`. That means `map` is available for many, many types. It also means you can write a function that uses `map`, and immediately be able to apply it to any of those types with the assurance that the result will be sensible (as defined by the functor laws).

For example:

```
negateAll : lifted number -> lifted number   | lifted of Functor
negateAll x =
  map negate x


> negateAll (Just 3)
Just -3 : Maybe number

> negateAll Nothing
Nothing : Maybe number

> negateAll [1, 3, 5]
[-1, -3, -5] : List number
```

But wait

There's something interesting here. In the chapter on monoids, I used this syntax for type classes:

```
 1 type class Monoid =
 2   requires
 3     append : ty -> ty -> ty   | ty of Monoid
 4     empty :  ty               | ty of Monoid
 5   checkWith
 6     ...
 7   provides
 8     concat : List ty -> ty    | ty of Monoid
 9     concat list = ...
10     ....
```

On lines 7 and following, I implied that `concat` and other functions are "contained" in `Monoid`, in much the way that methods in object-oriented languages are contained in classes.

But our `negateAll` is a free-standing function, not "inside" anything at all. In fact, that's always the case. If you look at the PureScript source for `Monoid`, you'll see a module structure like this:

```
module Data.Monoid ...

class Semigroup m <= Monoid m where
  mempty :: m

-- We're now outside the `class` definition

power :: forall m. Monoid m => m -> Int -> m
power x = ...
```

The provided functions just happen to be in the same module as the type class declaration. (Where else would you put widely-applicable `Monoid` functions?) But they didn't have to be.

I lied about provided functions in the `Monoid` chapter because I didn't want to throw too much at you all at once.

Text omitted

Metaphors

It's easy to think of `map` as transforming all the elements in a container.

- `(List Int)` is a container of zero or more `Int`s.
- `(Maybe Int)` is a container of either zero or one `Int`s.
- `(Result String Int)` is a container of either zero or one `Int`s.

But what about functions? or decoders? How are *they* containers?

> Alice laughed. "There's no use trying," she said: "one can't believe impossible things."
>
> "I daresay you haven't had much practice," said the Queen. "When I was your age, I always did it for half-an-hour a day. Why, sometimes I've believed as many as six impossible things before breakfast."
>
> – Lewis Carroll (Charles Dodgson) *Through the Looking Glass*

I'm not as practiced as Alice's Queen, but I can imagine a function as a container. My reasoning would follow these steps:

1. A `Maybe` can contain a value. `map` selects it (if present) and applies a function to it.

2. A `List` can contain many values. `map` selects them (if any) and applies a function to each.
3. A `(Function arg)` contains all the values it could ever produce. `map` selects the one corresponding to the `arg` and applies a function to it.

I'm probably good enough to come up with *some* argument that any functor you provide can be imagined to be a container.

However, it might be more useful to think about functors differently. Rather than emphasize how they're all the same ("container-like"), how about emphasizing their differences?

1. `Maybe` is special in that we know it's used in contexts where nothing happens, and nothing can be done to `Nothing`.
2. `List`s and `Array`s are used in contexts where there are potentially many values that `map` must keep in order.
3. Functions are used in contexts where you frequently want to compose them to create completely new functions.

In the jargon, you'll hear it said that functors provide a *computational context* that represents some form of active behavior that happens before a function is called and after it returns a value.[2]

For now, the difference between a passive container and an activist context isn't that important, but it can get interesting when you start "reifying" (making real) the context and manipulating it.

Using the context

As a brief example, let's look at a function very similar to `map`, called `apply`. It's from a type class called `Applicative` that's a slight extension of `Functor`. Here's `apply`'s type annotation, compared to `map`'s:

```
map  :        (a -> b) -> lifted a -> lifted b  | lifted of Functor
apply: lifted (a -> b) -> lifted a -> lifted b  | lifted of Applicative
```

The syntactic difference is that the function argument is itself lifted. For `Maybe` as an instance, we'd have this implementation:

[2]For a long time, I didn't understand what people meant by "computational context" and suspected that it was a synonym that just sounded important. (Like how people use the word "concept" because "idea" seems common.) It was a two-part blog post, "An Intuition on Context," by Matt Parsons that finally got it through my thick head. (Though I think my interpretation above is slanted somewhat differently than his, so don't blame him for any mistakes.)

```
apply : Maybe (a -> b) -> Maybe a -> Maybe b
apply maybeF maybeA =
  case maybeF of
    Just f -> map f maybeA
    Nothing -> Nothing
```

That seems unimpressive, but there's a big difference. Whereas `map` cannot make any use of the context, `apply` can. `map` can never turn a `Just` into a `Nothing` or alter the length of a list, but `apply` can. We'll eventually build on this to produce various useful behaviors.

For the moment, let me give you just a peek. `List` is an `Applicative` instance. Here's a PureScript-like example of using `apply` on two lists:

```
> apply [(+)1, (+)2] [10, 20, 30]
[11,21,31,12,22,32]
```

It happens that the implementation of `apply` for `List` applies each function in the first argument to each value in the second. (That's not required by the applicative laws, but it's consistent with another type class we'll see later.)

Another way to write that expression would be this:

```
> apply (map (+) [1, 2]) [10, 20, 30]
[11,21,31,12,22,32]
```

That's an expression that would have to be a nested loop in many languages.

Such a direct use of `apply` and `map` is awkward to read, but languages like PureScript provide operators to make it easier:

```
> (+) <$> [1, 2, 3] <*> [10, 20, 30]
[11,21,31,12,22,32,13,23,33]
```

The neat thing about that is how much it looks like normal function application:

```
> (+)      1          10
11
> (+) <$> [1...] <*> [10...]
```

You'll see this approach called the "applicative style." Here's how to add numbers lifted into the `Maybe` universe:

```
> (+) <$> Just 3 <*> Just 7
(Just 10)

> (+) <$> Just 3 <*> Nothing
Nothing
```

What now?

I’m deferring the next two chapters, so skip them and let’s look at some PureScript.

II Purescript

The main PureScript site is http://www.purescript.org.

Installation instructions are here: http://www.purescript.org/learn/getting-started/.

Note also the list of editor plugins.

This book was written using PureScript version 0.11.6.

After you've installed PureScript, also install a tool that makes compiler errors more readable.

```
npm install -g purescript-psa
```

11. Blundering into PureScript

If you are reading this in the free sample, be sure to look at the description of what the sample contains. It describes some prerequisite material (also included in the sample) that you probably want to at least skim.

There are a variety of syntactic and naming differences between Elm and PureScript. As far as semantics – behavior – goes, PureScript is *mostly* a superset of Elm.

If this book had stopped after the Elm material, one way to learn PureScript – the way I'd do it – would be to skim through the Elm chapters, trying to do the same things in PureScript. When things broke or behaved funny, I'd do some investigation. By repeating (roughly) that earlier sequence of learning, I'd expect PureScript ideas to get anchored more firmly in my brain.

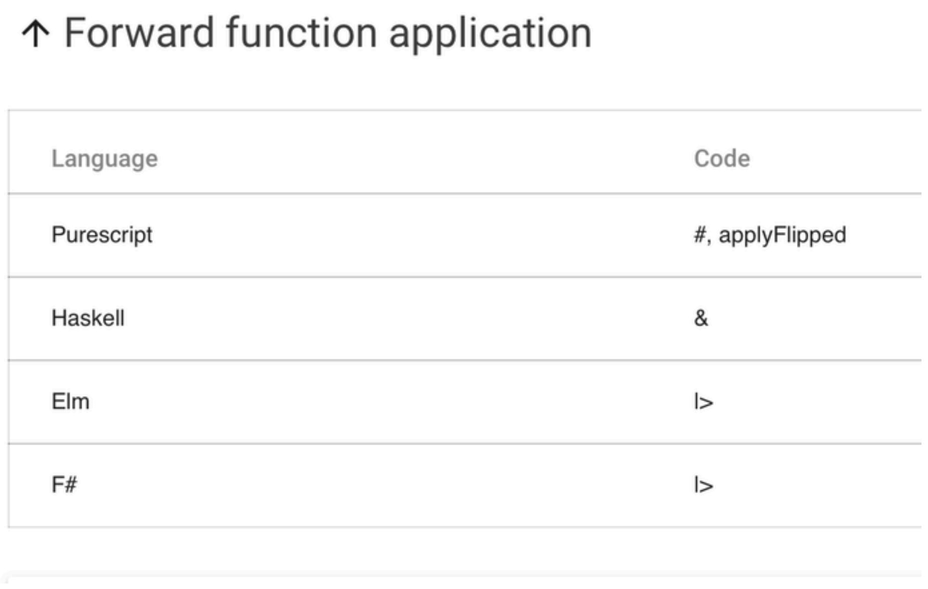
Also, I'd learn not just syntax and semantics – the stuff of a language manual – I'd also see how Elm idioms translate into PureScript and how my Elm programming habits and workflow should change for PureScript.

This chapter is an idealized and condensed version of the first part of such a journey. It highlights what I expect you to stumble over. Instead of making you search for explanations (like I did!), I provide them immediately.

The result is nothing like a complete PureScript reference. That's the PureScript Language Reference, supplemented by the (free) ebook *PureScript by Example*.

Unsurprisingly, those two sources don't mention Elm. For comparisons between the two languages that make it easier to find information you once learned but can't quite remember, see these:

- PureScript for Elm Developers, maintained by Marco Sampellegrini.
- The Functional Programming BabelFish, from Håkon Rossebø, compares vocabulary across languages, in both terse tabular form and an expanded form. The latter looks like this:


↑ Forward function application

Language	Code
Purescript	#, applyFlipped
Haskell	&
Elm	|>
F#	|>

Note: Elm and PureScript offer similar programming experiences to the expert, but Elm is more polished and tailored to the beginner. You may experience more early frustration than in your Elm programming; please be patient.

PureScript expects projects

This book's PureScript code is in the `purescript` subdirectory. PureScript strongly prefers that each app be in its own self-contained subdirectory. Within that, the source code is in `src/`, test code in `test/`, and the app's dependencies are listed in `bower.json`.

In this chapter, you'll be working with the `purescript/blunder` project. As with all project directories throughout the book, you'll have to start by downloading a whole bunch of packages:

```
% cd static-fp/purescript/blunder
% bower install
```

Starting the repl

Start the repl like this:

```
$ pulp repl
```

You'll see a huge list of `Compiling This.That.TheOther`. PureScript favors small modules. Admire the list, then ignore it.

Just before the repl prompt, you'll see this:

```
import Prelude
```

That marks an important difference from Elm. Every Elm module, and the repl, begins with an *invisible* statement like this:

```
import Basics exposing (..)
```

`Basics` includes functions like `(+)`, `(==)`, `(>>)`, `flip`, and `toString`. `Prelude` plays the same role, except that it's only automatically imported into `pulp repl`. Unless you specifically ask for it in a PureScript module, you won't get it. Even as simple a function as this:

```
add1 x = 1 + x
```

… would produce an error:

```
Compiling Scratch
Error:
  in module Scratch
  at src/Scratch.purs line 8, column 12 - line 8, column 13

    Unknown operator (+)
```

So most modules will start with this:

```
module Scratch where
import Prelude
```

The benefit of requiring an explicit `import Prelude` is that you can make your own Prelude, one that has exactly the names you want.[1]

Even given the Prelude, you'll probably be surprised by which modules you have to specifically import. For example, `Maybe` is not provided by `Prelude`:

```
> Just 3
Error:

  Unknown data constructor Just
```

You have to explicitly import it.[2] It's not in a module named `Maybe`, though. Conventionally, modules that hold a datatype are prefixed with `Data`:

[1]After the success of the Rails web framework, "convention over configuration" has become a popular guideline. The older static FP tradition leans toward configuration more than is perhaps popular today. Elm is an exception to that.

[2]You can put additional imports into `.pulp-repl`. I chose not to do that because it only applies to the repl, not to source files. If you do add to it, beware that the default `.gitignore` ignores the file. If you're working in a team, that's not good.

```
> import Data.Maybe
> Just 3
(Just 3)
```

Notice that `import` behaves differently than in Elm. By default, PureScript's `import` exposes all the names in the imported module. This is called *open import*. To get the equivalent effect in Elm, you'd do this:

```
> import My.Module exposing (..)
```

I'll cover PureScript's various forms of `import` later.

Trying out functions

Functions and operators are used just as in Elm:

```
> 1 + 2
3

> (+) 1 2
3

> negate 2
-2
```

Printing types

One surprise is that types aren't automatically printed. To see the type of a value or expression, you use the repl-specific `:type` command:

```
> :type 3
Int

> :type 1 + 2
Int

> import Data.String as String
> :t String.length
String -> Int
```

`:type` can be abbreviated `:t`.

Numbers

You’ve seen that PureScript has `Int`. Does it have `Float`?

```
> :type 3.3
Number
```

Like Elm, PureScript has to cope with the fact that JavaScript really only has floating point numbers. Elm gives you three types for numerical values:

- `Int`, which is your declaration to the compiler that the value could be represented as an exact integer.
- `Float`, your declaration that the number might or might not be an exact integer.
- `number`, a special-case pseudotype that allows you to mix integers and floats in expressions, so that you can write code like `1 + 3.2`.

These types are only relevant at compile time: during execution, everything’s a JavaScript number. The types serve to restrict what you can do with the numbers.

PureScript slices things differently. It has `Int`, a type with the same meaning as Elm’s version. `Number` is roughly equivalent to Elm’s `Float`: that is, it could *happen* to have an integral value, but it doesn’t have to. A big difference, though, is that `Int` and `Number` values can’t be combined with operators like `(+)`:

```
> 1 + 3.3
Error:
  Could not match type
    Number
  with type
    Int
```

(We’ll see much more about error messages in a later chapter.)

Why this restriction? Think of `Int` and `Number` as two separate instances of the same type class, in something like the way that both `List` and `Array` are instances of `Monoid`. You wouldn’t expect the `append` operator to allow you to combine an `Array` and a `List`, so – unless there’s special handling, like Elm’s – you don’t get a `(+)` operator that combines an `Int` and a `Number`.

To add an `Int` and a `Number`, you have to convert the `Int`:

```
> import Data.Int
> toNumber 1 + 3.3
4.3
```

Lists and arrays

In the functions chapter, the very first function I had you try was `List.take`. Let's try it with PureScript:

```
> import Data.List
> take 2 [1, 2, 3]
```

A long error message appears. The important part is this:

```
  Could not match type
    List
  with type
    Array
```

The expression `[1, 2, 3]` is an `Array`, not a `List`, which we can see like this:

```
> :t [1, 2, 3]
Array Int
```

This is a way in which PureScript breaks with tradition more than Elm does. Elm sticks with using a linked list as its fundamental data type. PureScript instead follows JavaScript in using the array as the fundamental type.[3]

So we'll need to use `Data.Array`'s version of `take`, not `Data.List`'s:

[3]PureScript arrays are in fact exactly JavaScript arrays. That makes passing data between PureScript and JavaScript easier than it is in Elm. (Even Elm's `Array` is not a JavaScript array.) Because changing one element of an array requires copying the whole thing, PureScript's choice can be a performance problem. Just as in Elm, you might want to do most of your intermediate work with lists. (Performance is the reason that Elm's `Array` is a non-array data structure with an array-like interface.)

```
> import Data.Array
> take [1, 2, 3]

  Conflicting definitions are in scope for value
  `take` from the following modules:
    Data.Array
    Data.List
```

Darn it! I want to have only open-imported one of them. I'll exit the repl and then redo some work:

```
> :quit
See ya!

$ pulp repl
PSCi, version 0.11.6
Type :? for help

import Prelude
>
```

Redoing and undoing in the repl

Using `:quit` and restarting is the most thorough way to get back to a clean repl state. On Unix and Macs, you can also quit by typing the `Control-D` character or banging repeatedly on `Control-C`. I don't know if that works on Windows.

The `:clear` command undoes all module imports, and also all name bindings from `(<name> = <calculated value>)` expressions. Unfortunately, one of the module imports it undoes is `Prelude`, so you have to manually `import Prelude` after that.

The `:reload` command is like `:clear`, but it then reloads all imported modules (including `Prelude`). When you start putting code in files, you'll reload changed modules with `:reload`.

Of these commands, you'll use `:reload` the most.

Unlike Elm, re-importing a changed file does not recompile it, so the changes aren't visible until `:reload`. If you're as habit-bound as I am, that will result in several confusing episodes until you've retrained your brain.

Because `:reload` destroys all previous name bindings, this sequence of events doesn't work like it does in Elm:

```
> import Scratch
> sample = 3838
> scratchFunction sample 3
3841                        -- That's the wrong result
                            -- Edit Scratch.purs
> :reload
Compiling Scratch
> scratchFunction sample 3
Error:
  Unknown value sample      -- `sample` has been erased
```

For my repl work, I usually put just-trying-it-out bindings like `sample` in the module I'm working on. That way, they get recreated as the module is reloaded.

Back to resolving the conflict. In the new repl, I'll alias the modules:

```
> import Data.List as List
> import Data.Array as Array
```

And now `Array.take` works as we'd expect:

```
> Array.take 1 [1, 2, 3]
[1]
```

There's no special syntax for creating true `List`s, so they're as awkward to work with as Elm `Array`s are. You create them with function calls and constants.

```
> import Data.List -- open import is more convenient for remaining examples
> singleton 1
(1 : Nil)
```

`Nil` represents the empty list. You can put elements on the front of a list:

```
> 1 : Nil
(1 : Nil)
> 1 : 2 : Nil
(1 : 2 : Nil)
```

Notice that the "cons" operator is `(:)` instead of Elm's `(::)`.

A more convenient way of creating a list is to convert an array:

```
> fromFoldable [1, 2, 3]
(1 : 2 : 3 : Nil)
```

Notice the function is named `fromFoldable`. That signals it will work with any type that supports a `foldl` operation (that is: any instance of the `Foldable` type class that you'll learn about later). So you can use the same function to create lists from lots of types, such as `Maybe`:

```
> import Data.Maybe
> fromFoldable (Just 3)
(3 : Nil)
```

Defining functions

Function definitions look a lot like in Elm:

```
> plus1 x = x + 1
> plus1 3
4
```

You can also use point-free style:

```
> plus2 = (+) 2
```

Notice that I defined a new `plus2` instead of redefining `plus1`. That's because I wanted to save this surprise until now:

```
> plus1 = (+) 1
Error:
  The value plus1 has been defined multiple times
in value declaration plus1
```

You get to bind a name exactly once. So you can't do this either:

```
> x = 1
> x = 2
Error:
  Multiple value declarations exist for x.
```

If you want to "rebind" a value, you have to use `:reload` to make the repl forget *all* bindings.

Given what I've *just said* about "you get to bind exactly once," the following is surprising:

```
> f x y = x + y
> f 1 2
3

> f x y = x * y + 333
```

Why was I allowed to redefine the function? Just to add to the confusion, look at this:

```
> f 1 2
3
```

Wait…? It didn't actually get redefined? What's up?

This is a consequence of PureScript's function definitions allowing a richer form of argument list pattern matching than Elm's. In Elm, we've seen a lot of code that looks like this (from the `Maybe` module):

```
withDefault : a -> Maybe a -> a
withDefault default maybe =
    case maybe of
      Just value -> value
      Nothing -> default
```

In PureScript, you can split the cases between what seem to be two functions with the same name:

```
> withDefault default (Just value) = value
> withDefault default Nothing = default
> withDefault 7 (Just 3)
3

> withDefault 7 Nothing
7
```

The compiler essentially synthesizes a single function from the two definitions. Picture the result like this:

```
withDefault arg1 arg2 =
  case (arg1, arg2) of
    (default, Just value) -> value
    (default, Nothing) -> default
```

Notice that implies the patterns are checked in order.

You can use "don't care" patterns just as you can with `case`. So the definitions could have looked like this:

```
> withDefault    _      (Just value) = value
> withDefault default        _       = default
```

That's handy for showing which parameters matter in which case.

Now suppose I hadn't typed the second version of `withDefault`, but just this:

```
> withDefault  _  (Just value) = value
```

The compiler is fine with this, even though it doesn't handle all possibilities. Can we make the compiler be more demanding? And what happens if you give the function a case it doesn't handle? The answer is… don't do that (for now). I'll explain your options later.

With all that background covered, let's return to the original puzzle:

```
> f x y = x + y
> f 1 2
3

> f x y = x * y + 333
> f 1 2
3
```

What looks like a redefinition of `f` actually adds another case to the existing definition, producing something like this Elm code:

```
f arg1 arg2 =
  case (arg1, arg2) of
    (x, y) -> x + y
    (x, y) -> x * y + 333
```

The first case will match any arguments (binding the values to `x` and `y`), meaning the second case can never be reached. (Elm produces error messages for redundant cases; as of this writing, PureScript's repl doesn't. You will, however, get a warning if you compile outside the repl with `pulp build`. We'll see more about that later.)

Anonymous functions

The notation is the same as Elm's:

```
> (\x -> x - 1) 3
2
```

There's also a more concise form of the above:

```
> (_ - 1) 3
2
```

Note that eliminates the need to fiddle with `flip`, as we've occasionally done in Elm:

```
> (flip (-) 1) 3
2 : number
```

So that's a nice win.

Unlike its equivalent in some languages, _ isn't a general purpose shorthand you can use to create arbitrary functions. It only works with certain expressions. This page lists them all. I'll introduce more of them as they become relevant.

Type signatures

Unlike Elm, you can add type annotations at the repl, but there’s a wrinkle:

```
> withDefault :: Int -> Maybe Int -> Int
Error:
  The type declaration for withDefault should be
  followed by its definition.
```

Just as with Elm, the type annotation has to be followed by a definition of the function, but – given line-by-line evaluation in the repl – we never get the chance to type it. We need a way to evaluate lines as a group.

PureScript has a nice way of doing that, the repl’s `:paste` command:

```
> :paste
… withDefault :: Int -> Maybe Int -> Int
… withDefault     _     (Just value) = value
… withDefault default       _        = default
… ^D
```

The `^D` represents the `Control-D` character, which exits “paste mode.” The nice thing about `:paste` is suggested by its name: you can copy code from a source file and paste it into the repl without having to edit lines to add backslashes.

Type annotations look like Elm’s, except they use a double-colon instead of a colon. (As you saw earlier, the single colon is used to add an element to the front of a list.)

Type variables

Up until now, I’ve been careful to use concrete types in annotations. Let’s look at functions that work with quantified types. Here’s one:

```
> pack x y = [x, y]
```

What’s its type signature?

```
> :t pack
forall t2. t2 -> t2 -> Array t2
^^^^^^^^^^
```

That looks almost like an Elm type signature, except for the highlighted part. PureScript requires type variables to be “declared” before they’re used. Here’s the kind of type annotation you’d put in a module:

```
> :paste
… pack :: forall a. a -> a -> Array a
… pack x y = [x, y]
```

Don’t forget the period after the declaration! If you don’t, you’ll see this:

```
> :paste
… pack :: forall t t -> t -> Array t
… pack x y = [x, y]
… ^D
(line 1, column 20):
unexpected ->
expecting identifier or .
```

Type classes and type signatures

Earlier in the book, I described (using made-up syntax) how PureScript allows particular function names to be used with many types, so long as the types are all instances of the same type class. Compare this Elm code:

```
> Maybe.map negate (Just 3)
Just -3 : Maybe number
> List.map negate [1, 2]
[-1,-2] : List number
```

… to this PureScript code:

```
> map negate (Just 3)
(Just -3)

> map negate [1, 2]
[-1,-2]
```

What’s the type of `map`?

```
> :type map
forall a b f. Functor f => (a -> b) -> f a -> f b
              ^^^^^^^^^^^^
```

Look first at the ^^^^^^^^ code. That `f` is the equivalent of the `lifted` in my earlier not-a-real-language declaration of `map`:

```
map : (a -> b) -> lifted a -> lifted b    | lifted of Functor
```

`lifted` and `f` aren't type variables,[4] but PureScript asks us to declare them as if they were:

```
> :type map
forall a b f. Functor f => (a -> b) -> f a -> f b
^^^^^^     ^^
```

Once you're used to such type signatures, they quickly signal that "this function works on any data that is a functor."

`map` is declared in `Data.Functor`, but provided (re-exposed or re-exported) by `Prelude`. `Prelude`, though, doesn't provide all of `Data.Functor`'s names. For example, it doesn't provide `Data.Functor.mapFlipped`. If you want that, you have to import it:

```
> import Data.Functor
> :t mapFlipped
forall f a b. Functor f => f a -> (a -> b) -> f b
```

If you want to know what's in the Prelude, you can look at some documentation. The PureScript documentation is generally harder to use than Elm's, so I provide a later chapter to help you answer the question "what in PureScript is like this Elm function?"

Now let's look at the functions mentioned in the monoids chapter. Here's `append`'s type:

```
> :t append
forall a. Semigroup a => a -> a -> a
```

Surprisingly, the type class is `Semigroup` rather than `Monoid`. PureScript tends to be highly modular, dividing concepts into small constituent parts. It builds the type class `Monoid` out of a simpler `Semigroup`.

In PureScript, a `Semigroup` must have an associative `append` but doesn't require an empty element. A `Monoid` is a `Semigroup` that *does* require an empty element, named `mempty` (think "monoidal empty").

It happens that the `Prelude` exposes `Semigroup` but doesn't expose `Monoid`. So if you want `mempty`, you have to ask for it:

[4]At least, they're not type variables of the same kind as those we've seen before. I don't want to get into that now.

```
> import Data.Monoid
> :t mempty
forall m. Monoid m => m

> append [1] mempty
[1]
```

Warning: If you're like me, you'll frequently type `mempty` as `empty` and then get puzzled why this type you're *sure* is a monoid doesn't have an empty element.

Note: Elm aliases `append` to the operator `(++)`. In PureScript, it's `(<>)`:

```
> [1, 2] <> [3, 4] <> [5, 6]
[1,2,3,4,5,6]

> "hi " <> "there " <> "Dawn"
"hi there Dawn"

> Just [3] <> Just [4]
(Just [3,4])
```

Exercises

In order to get some practice reading PureScript documentation, develop the beginnings of a feel for what you're likely to find where, and get your hands dirty at the repl, examine these module documents:

Data.List
Data.Array
Data.Monoid
Data.Functor

Until you get bored, create a repl example of each of the functions.

Oh pretty boy, can't you `Show` me nothing but surrender?[5]

If you're following along, you've probably mistyped something and seen an error message about the dreaded (by me, at least) `Data.Show.Show` type class. For example, if you're used to querying the type of a function like this in the Elm repl:

[5]From the title track of Patti Smith's album "Horses," certainly one of the top five rock-and-roll albums of all time.

```
> List.map
<function> : (a -> b) -> List a -> List b
```

… your habit is setting you up for a world of annoyance in PureScript:

```
> map
Error:
in module $PSCI
at <internal> line 0, column 0 - line 0, column 0

  No type class instance was found for

    Data.Show.Show ((t3 -> t4) -> t5 t3 -> t5 t4)

  The instance head contains unknown type variables.
  Consider adding a type annotation.

while applying a function eval
  of type Eval t2 => t2
                     -> Eff
                          ( console :: CONSOLE
                          )
                          Unit
  to argument it
while checking that expression eval it
  has type Eff t0 t1
in value declaration $main

where t0 is an unknown type
      t1 is an unknown type
      t2 is an unknown type
      t5 is an unknown type
      t4 is an unknown type
      t3 is an unknown type
```

The important bit is this:

```
  No type class instance was found for

    Data.Show.Show ...
```

PureScript's equivalent of `toString` (called `show`) *only* works for types that are instances of the `Show` type class. (That may change in future versions of PureScript.)

Functions (type `Function a b`) are not instances of `Show`.

Functions aren't the only values that aren't instances of `Show`. Sum types that you create are not. I'll describe shortly how you can make them `show`, but let's first look at sum types in general.

Sum types

Sum types are the same as in Elm, with the exception of a different keyword (`data` instead of `type`):

```
> data MyType wrapped = First wrapped | Second
> one = Second
> two = First 3
```

By binding the values to names, I prevented the repl from trying to display them. It can't because `MyType` isn't an instance of `Show`:

```
> Second
Error:
  No type class instance was found for
    Data.Show.Show (MyType t3)
```

Phantom types and type aliases

PureScript's sum types don't have to have any value constructor:

```
data PercentTag
```

You can't create values of such types, but you can use them alongside phantom types for tagging. Here's a tagging type:

```
> data Tagged tag value = Tagged value
```

We can now write a `percent` function that creates a tagged `Number` and an `unwrap` function that extracts the wrapped value:

```
> :paste
… percent :: Number -> Tagged PercentTag Number
… percent = Tagged
…
… unwrap :: Tagged PercentTag Number -> Number
… unwrap (Tagged n) = n
… ^D

> :t percent 33.3
Tagged PercentTag Number

> unwrap (percent 33.3)
33.3
```

That's all the same as in Elm. PureScript has some additional simplifications and improvements, but I'll leave them for a later chapter.

We probably don't want to keep typing `Tagged PercentTag Number` all over the place, so we can use PureScript's equivalent of `type alias`, which – confusingly – is named `type`:

```
type Percent = Tagged PercentTag Number

percent :: Number -> Percent
percent = Tagged

unwrap :: Percent -> Number
unwrap (Tagged n) = n
```

Implementing `Show`

Let's make `show` work for a sum type, working up to it in stages. You can find the finished product in `purescript/blunder/src/Show.purs`.

Consider first a simpler type that uses no type variables:

```
data MyType = First | Second
```

Here's a `Show` instance:

```
instance showMyType :: Show MyType where
  show First = "First"
  show Second = "Second"
```

The first line declares we're creating an instance of `Show` for the type `MyType`, and the instance is named `showMyType`. The name isn't used in our code, but it's used to make the generated JavaScript more readable (which is a core value for PureScript).

The remaining two lines define the `show` function for `MyType`. Notice that we use the trick of defining a different function body for each case.

Now let's redefine `MyType` (in the repl, this would require a `:reload`) to add a type variable:

```
data MyType wrapped = First wrapped | Second
```

We can make `show` work with two simple substitutions:

1. `MyType` becomes `(MyType wrapped)`, and
2. `First` becomes `(First _)`.

Like this:

```
instance showMyType :: Show (MyType wrapped) where
  show (First _) = "First"
  show Second  = "Second"
```

That's pretty unsatisfactory. Surely we want to `show` the value `First` wraps, like this:

```
  show (First x) = "(First " <> show x <> ")"
```

If we make that change, the code won't compile:

```
  No type class instance was found for
    Data.Show.Show wrapped1
```

It makes no sense to `show` a `First wrapped` unless the `wrapped` value can also be shown. That constraint has to be made explicit:

```
instance showMyType :: Show wrapped => Show (MyType wrapped) where
                       ^^^^^^^^^^^^^^^
```

That should remind you of how functions like `mapFlipped` depend on `Functor`:

```
> :t mapFlipped
forall f a b. Functor f => f a -> (a -> b) -> f b
              ^^^^^^^^^^^^
```

`mapFlipped` only works when the first argument's type is a `Functor`, and `showMyType` only works when the wrapped type is a `Show`.

Here, then is the final instance declaration:

```
data MyType wrapped = First wrapped | Second
instance showMyType :: Show wrapped => Show (MyType wrapped) where
  show (First x) = "(First " <> show x <> ")"
  show Second = "Second"
```

So that's how we can print a `First` value:

```
> show (First 33)
"(First 33)"
```

And it works for the `Second` case:

```
> show Second
Error:
  The inferred type
    forall t2. Show t2 => String
  has type variables which are not mentioned in the body of the type.
  Consider adding a type annotation.
```

... Oops. Do you understand what went wrong? (I couldn't figure it out.)

Adding type annotations to values

The `showMyType` instance defines a single function, `show`, that could also have been written like this:

```
instance showMyType :: Show wrapped => Show (MyType wrapped) where
  show value =
    case value of
      First x -> "(First " <> show x <> ")"
      Second -> "Second"
```

The type of that `show` *instance* has to match the *type class's* polymorphic function `show`. What's the latter's type?

```
> :t show
forall a. Show a => a -> String
```

Read that as: any `Show` type can be converted into a string.

What's the instance's type? We have no way of looking at it, actually. If we did, we would see it narrowed like this:

```
"show (MyType instance)" :: forall wrapped.
    Show wrapped => MyType wrapped -> String
```

Read that as: a type and its `MyType` wrapper can be converted into a string, provided the type is an instance of `Show`.

So what's the type of a plain `Second`, which doesn't actually wrap anything?

```
> :t Second
forall wrapped. MyType wrapped
```

The type signature doesn't restrict `wrapped` at all. It might be an instance of `Show`, but it might not. Lots of values aren't, after all. So the compiler refuses the match.

The compiler follows its mechanical algorithm even though we know it doesn't need to. You may have seen that sort of stubbornness in other programs.

Giving the compiler what it needs

We could get cute and force the compiler to narrow the type of a particular use of `Second`. Unlike the above, this particular use of `Second` can be stringified:

```
> [First 3, Second]
[(First 3),Second]
```

Because the compiler knows the first element is a `MyType Int`, and because it knows that all elements of an array must be of the same type, it infers that the second element is also a `MyType Int`. It also knows that `Int` is indeed an instance of `Show`, so it's happy.

Yay.

Still, it would be nice to show a standalone `Second`. You do that by forcing the compiler to narrow its type:

```
> Second :: MyType Int
Second
```

Why does Elm work?

Elm is fine showing values with quantified types:

```
> Second
Second : Scratch.MyType a
```

It can do that because `toString` is a special case, implemented in JavaScript. It can do things Elm code can't do. However, that also means that you can't choose how your own type should be shown.

Modules and imports

PureScript modules are stored in the `/src` subdirectory. Unlike Elm, there's no required relationship between the file name and the module name, except that the file must end in `.purs`. So a file named `Name.purs` can contain a module `DifferentName` or even `Different.Name`.

In such a case, you'd `import` the module name (`Different.Name`) instead of the file name. Fortunately for everyone's sanity, PureScript's naming convention is the same one that Elm enforces. That is, `Different.Name` would be found in the file `src/Different/Name.purs`.

When the repl starts, it will compile all PureScript files in `src`. If it encounters a compiler error, it stops:

```
$ pulp repl
at src/Blunder/First.purs line 22, column 1 - line 22, column 1

Error:
  Unable to parse module:
  unexpected end of input
582 $
```

This compile-all approach allows the repl to know what modules are available even if the file names and module names don't correspond.

As a side effect of that, a *new* file won't automatically be visible to the repl:

```
$ pulp repl
Compiling Other
PSCi, version 0.11.6
Type :? for help

import Prelude
> -- Create `Other2.purs` here
> import Other2
Error:
  Unknown module Other2
```

To force a compilation (and the resulting visibility), use `:reload`:

```
> :reload
Compiling Other2
> import Other2
>
```

A module definition

A module starts like this:

```
module Scratch where
```

Note the `where`.

By default, a module exposes all its names. The mechanism for exposing only some of the names is like Elm's:

```
module Scratch
  ( intList
  , cases
  , MyType(..)  -- a sum type and its constructors
  ) where
```

… the only difference being that there's no `exposing` keyword before the parenthesized list.

Imports follow the module declaration:

```
import Prelude

import Data.List (List) as List
import Data.Array
import Data.Tuple
```

The rest of the module is a list of bindings (that is, constant and function definitions):

```
intList :: List Int
intList = List.singleton 555

increment :: Int -> Int
increment i = i + 1
```

Exercises

Work with purescript/blunder/src/Scratch.purs:

```
% pulp repl
...
> import Scratch
```

1. Check the types of `intList` and `tuple`.

   Note: there's no literal syntax for tuples. Also, tuples always have only two elements. PureScript seems to lean more toward representing even unimportant data with records.
2. Check the type of `intArray`. How does the compiler know that's the type?

   Hint: look at the imports.
3. Type `Second :: MyType Int` to the repl 300 times. That's my estimate of how many times it took me before I remembered the need to add the `Int`.
4. Change how `(First x)` gets shown. (Perhaps remove the parentheses.) Confirm that your change worked.
5. Try using `cases`.
6. Try adding a case for `8 8` in different places. (Are you warned of unreachable cases?) Discussion of this exercise's results appears below.

`import` variants (reference)

Here are the differences between Elm's `import` and PureScript's, with Elm's version first. The commentary that follows explains the PureScript version.

```
import X.Y exposing (..)
------------------------
import X.Y
```

- PureScript exposes everything unless you tell it not to.
- You can't use the fully-qualified name. That is, `X.Y.foo` wouldn't work.

```
import Maybe exposing (isJust)
------------------------------
import Data.Maybe (isJust)
```

- `isJust` will be available, but no other names from `Data.Maybe`.
- In particular, the value constructor `Just` won't be available.

```
import Maybe exposing (Maybe(..))
---------------------------------
import Data.Maybe (Maybe(..))
```

- Importing a type and its constructors works as it does in Elm.
- You can replace the `(..)` with a comma-separated list of constructor names.
- To import a type class, you use its name, prefixed by `class`: `(class Show)`.

```
----------------------------------
import Data.Map hiding (singleton)
```

- You can import everything *except* a list of names. This is useful when two modules bind the same name.
- There's no way to do the equivalent in Elm.

```
----------------------------------
import Data.Tuple (fst) as Tuple
```

- `fst` is the only name imported from `Data.Tuple`. The `snd` function, for example, is unavailable.
- A use of the imported name has to be qualified: `Tuple.fst`.
- There's no way to do the equivalent in Elm.

Import style

Elm style makes heavy use of `import ... as`:

```
import Json.Decode as Decode exposing (Decoder)
```

That's both to avoid name clashes (the same name defined in two imported modules) and also to make it easy to guess where a particular function is defined.[6] Elm names often seem to be chosen with the assumption that they'll be module-qualified in use. For example, `succeed` would be an uninformative name on its own, whereas `Task.succeed` is better. I follow that convention, so much of my code expects client code to use `Animal.named "fred"` instead of just `named "fred"`.

PureScript style leans toward exposing all – but only – the names you'll be using in the module:

```
import Data.List (List, singleton)
import Data.Array (snoc)
import Control.Monad.State (State, state, execState)
```

Naming everything you'll use greatly reduces the chance of accidental name clashes, and it saves typing. PureScript is opinionated about this style. If your module has imports that don't list specific functions, `pulp build` will give you warnings of this form:

```
[2/5 ImplicitImport] src/Scratch.purs:8:1

  8  import Data.Array
     ^^^^^^^^^^^^^^^^^

  Module Data.Array has unspecified imports, consider using
  the explicit form:

    import Data.Array (singleton)
```

As with Elm's compiler, each warning gives you the code that will make it go away.

In this book, I'll lean more toward a module-qualified style than idiomatic PureScript code does. I just like it better. However, the ubiquity of type-class-based polymorphism means there will be a lot of free-floating `map`s and `foldl`s and the like.

Total and partial functions

This, from Scratch, is called a *total function*:

[6] In a perfect world, we'd all be issued an editor at birth that understands all programming languages, even the newest, as well as their compilers do – and that can hide module names when we don't need them and show them, quickly and unobtrusively, when we do. Meanwhile, in our world…

```
cases :: Int -> Int -> Int
cases 0 0 = 10
cases 0 1 = -1
cases _ 3 = -3
cases 8 _ = -8
cases x y = x + y
```

“Total” means that the function can handle every input the compiler will allow you to give it.

If you remove the last line, you’ll get an error message like this:

```
> import Scratch
> :re
Error:

  A case expression could not be determined to cover all inputs.
  The following additional cases are required to cover all inputs:

    _ _

  Alternatively, add a Partial constraint to the type of the
  enclosing value.
```

What the last sentence asks for is this type annotation:

```
cases :: Partial => Int -> Int -> Int
```

`Partial` is a type class like `Monoid` or `Functor`. You don’t want an instance of it; it exists to fail *because* there’s no instance. That is, here’s what happens when you try to use `cases`:

```
> cases 3 5
Error found:
  No type class instance was found for

    Prim.Partial
```

A function that you can define but not use would not be… useful, so there’s a special function that converts such a *partial function* to one the compiler won’t complain about:

```
> :t cases
Partial => Int -> Int -> Int

> :t unsafePartial cases
Int -> Int -> Int
```

Can you guess the type of `unsafePartial`?

```
> :t unsafePartial
forall func. (Partial => func) -> func
```

The function produces a function stripped of the `Partial` type class constraint.

Tricksy

`unsafePartial` can't be implemented in PureScript. Instead, the compiler is faked out by sending the function argument into JavaScript:

```
foreign import unsafePartial :: forall a. (Partial => a) -> a
```

The compiler has no choice but to believe that the type annotation is true.

The JavaScript just returns the function it got:

```
exports.unsafePartial = function (f) {
  return f();
};
```

So `unsafePartial` is the way you call partial functions in PureScript:

```
> unsafePartial cases 8 3
-3
```

It represents a pinky promise[7] that this use of `cases` will never be given values it can't handle.

What happens if you break your promise? This:

```
> unsafePartial cases 1 8
.../node_modules/Scratch/index.js:73
   throw new Error("Failed pattern match at Scratch line 29,
   column 1 - line 29, column 38:
   " + [ v.constructor.name, v1.constructor.name ]);

Error: Failed pattern match
    at .../node_modules/Scratch/index.js:73:23
    at .../node_modules/Scratch/index.js:74:15
    at Object.<anonymous> (.../node_modules/$PSCI/index.js:8:28)
    at Module._compile (module.js:541:32)
```

[7]A pinky promise is a very serious promise, often made by intertwining your pinky (the farthest finger from the thumb) with that of the person to whom you're making the promise.

Admit it: because you've spent so much of your life looking at exception stack traces, you've been missing them. `unsafePartial` gives you a way to get them back.

`unsafePartial` is arguably useful in cases where the compiler forces you to handle impossible cases. If you're familiar with the "throw an exception and rely on the top level of the program to handle it" style of error handling, PureScript supports it.

That can be a reasonable design choice, though not exactly in the spirit of functional programming. I'll avoid partial functions in the rest of the book.

Pipelines

Pipelines of the Elm sort are less common in PureScript, but it has equivalents to all the pipeline operators.

`(|>)` is written `(#)`:

```
> 5 # negate # (+) 5
0
```

(I think those would be clearer written in Elm style, with each `(#)` on a separate line. I'm doing it as a single line because I'm working in the repl. Also: using vertical space seems less common in PureScript programs than in Elm programs.)

`(<|)` is written `($)`:

```
> Just $ 5 + 3
(Just 8)
```

`($)` seems more common in PureScript than `(<|)` is in Elm.

`(>>)` is written `(>>>)`:

```
> f = negate >>> (+) 5
> f 5
0
```

`(<<)` is written `(<<<)`:

```
> f = not <<< isJust
> f Nothing
true
```

A `Maybe` pipeline can be written using `map`. First, a bit of preparation:

```
> import Data.Maybe
> import Data.Array

> empty = [] :: Array Int
```

The explicit type for `empty` is because I want to map `negate` over it. Without `:: Array Int`, the compiler would object that the type of `empty` is an `Array` of anything, and `negate` only works with specific types.

Now, `map`:

```
> head empty # map negate # map ((+) 3)
Nothing

> head [3] # map negate # map ((+) 3)
(Just 0)
```

You'll see other ways of pipelining in the next chapter.

Records

Elm and PureScript have very similar record types.

Creation

Like Elm, a record literal defines its own type:

```
> point = {x : 5, y : 5.5}
> :t point
{ x :: Int
, y :: Number
}
```

Note the use of a single colon `(:)` between field name and value, unlike Elm's `(=)`. The types within the record are prefixed with `(::)`, as are all PureScript types.

By default, `show` isn't defined for record types. Even though both the fields in the record above can be shown, the whole record can't be.

In Elm, it's common to calculate the value of the field, bind it to a name, then use the name to initialize a field of the same name in the record. See line 2 below:

```
1  > seconds = 5 * 60 * 60
2  > record = { seconds = seconds, tag = "time" }
3  { seconds = 18000, tag = "time" } : { seconds : number, tag : String }
```

The same is true of PureScript, but there's a shorthand when the field name and bound name are the same:

```
> record = { seconds, tag : "time" }
> record.seconds
18000
```

As in Elm, it's common to alias the detailed type:

```
> type Point = {x :: Int, y :: Number}
```

Unlike Elm, PureScript doesn't give you a value constructor with the alias:

```
> Point 1 3.3
Error:
  Unknown data constructor Point
```

You can create a constructor function in the normal way:

```
point :: Int -> Number -> Point
point x y = { x : x , y : y }
```

....but there's also a point-free shorthand:

```
point :: Int -> Number -> Point
point = { x : _ , y : _ }

> :t point 3 3.3
{ x :: Int
, y :: Number
}

> (point 3 3.3).y
"3.3"
```

getters

Fields are accessed in the usual way, and the dot notation can be chained:

```
> {x : { y : { z : 5 } } }.x.y.z
5
```

Accessor functions (like the freestanding `.x` function you'd get in Elm) don't come for free, but they're easily written using the anonymous function shorthand:

```
> map _.x [{x : 1} , {x : 5} , { x : 3}]
[1,5,3]
```

Pattern matching

You can pattern match on fields

```
> addPoint {x, y} = x + y
> addPoint {x : 5, y : 10, color : "red"}
15
```

Note that, as with Elm, you can omit fields from the pattern (in this case, `color`)

The syntax for matching both fields and the whole record is different (line 3):

```
> import Data.Tuple

> show point@{color} = Tuple (addPoint point) color

> show {x : 5, y : 10, color : "red"}
(Tuple 15 "red")
```

Simple update

Simple record updates have a syntax similar to Elm's:

```
> times10 rec = rec { x = 10 * rec.x , y = 10 * rec.y }
> addPoint $ times10 {x : 1, y : 3}
40
```

Note that update uses (`=`) instead of creation's (`:`).

In Elm, the record to be updated must be named. The following is not allowed:

```
{ { x = "new" } | x = "old" }
```

In PureScript, the value being updated can be a literal:

```
> ({x : "old"} {x = "new"}).x
"new"
```

… or it can be the result of a function, as on line 2 here:

```
1  > make v = { x : v , y : 2 * v }
2  > r = (make 5) { y = 88 }
3  > r.y
4  88
```

… and so on.

Like Elm, you can't add a field to a record with update notation:

```
> {x : 3 } { y = 3 }
Error:
  Type of expression lacks required label y.
```

Note that PureScript tends to call record fields *labels*.

Nested update

You can update nested fields without having to destructure and restructure (or use a lens):

```
> one = { x : {y : {z : 10}}}
> two = one { x { y { z = 3}}}
> two.x.y.z   -- `show` doesn't automatically work on records.
3
```

You can update more than one value at once, even at different levels:

```
> :r
> one = { x : 3, nested : { x : 5 }}
> two = one { x = 33 , nested { x = 888 }}
> two.x
33

> two.nested.x
888
```

Update functions

As with Elm, there are no freestanding update functions, but you can define your own:

```
> setX newVal rec = rec { x = newVal }
> (setX "new" { x : "old" }).x
"new"
```

There's a shorthand form for that:

```
setX = _ { x = _ }
```

Unfortunately, that makes the record the first argument, which fits poorly with pipeline style. You could `flip` the function:

```
> setX = flip _ { x = _ }
```

... though I rather think you'd be better off with the more wordy form.

Extensible records

Suppose you have this type:

```
> type Point = { x :: Number , y :: Number }
```

… and this function:

```
> :paste
… f :: Point -> Number
… f {x,y} = x + y
… > ^D
```

… you'll get a type error if you try to use a record with an extra field:

```
> f {x : 3.3 , y : 4.3 , color : "red"}
Error:
  Type of expression contains additional label `color`.
```

The notation for an extensible record is, again, similar to Elm's:

```
> type Point record = { x :: Number , y :: Number | record }
                ^^^^^^                                ^^^^^^^^
```

And now the function will work:

```
f :: forall whole . Point whole -> Number
f {x,y} = x + y

> f {x : 1.1 , y : 2.2 , color : "red" }
3.3000000000000003
```

Creating a new record type from an old one rather surprisingly uses parentheses:

```
type ColorfulPoint = Point (color :: String)
                           ^^^^^^^^^^^^^^^^^
```

That's because records are an instance of a more general mechanism that we'll see later.

`ColorfulPoint` is no longer extensible, because there's no type variable to match with further field descriptions. This version can be further extended:

```
> type ColorfulPoint2 record = Point (color :: String | record)
                      ^^^^^^                              ^^^^^^^^
```

`Result` is `Either`

Traditionally, the sum type Elm calls `Result` is called `Either`. `Err` is called `Left` and `Ok` is called `Right`:

```
> import Data.Either

> isLeft $ Left 3
true

> isRight $ Right 5
true
```

I used the `isLeft` and `isRight` functions because you can't `show` either a `Left` or `Right` value without telling the compiler of the type of the nonexistent alternative:

```
> Right 5 :: Either String Int
(Right 5)
```

As with `Result`, `map` works on the `Right` value:

```
> (map negate $ Right 5) :: Either Int Int
(Right -5)

> (map negate $ Left 5) :: Either Int Int
(Left 5)
```

For that reason, `Right` is typically used to wrap success values and `Left` error values.

What now?

Elm gives you only one way to side-effect the outside world or to retrieve changing information (like random numbers or the current time): return a `Cmd` to the Elm runtime.

PureScript and similar languages also access the outside world by constructing data. However, they do that in a way that's more fine-grained than Elm's, a way that can look more like the IO you're used to.

Appendices

Glossary

Italicized phrases correspond to other entries in the glossary. At some point, they'll become links.

abstract type:
: A type whose definition contains *type variables*. I use the term *quantified type* instead.

alias (module):
: Replacing one module name with another, typically shorter.

```
import Very.Long.Name as Name
```

alpha conversion:
: A fancy way of saying that you can freely rename a function's parameters and their uses within the body. The following two are equivalent:

```
f  n  =  n  +  n
f nnn = nnn + nnn
```

It's a bit tricky to get this right if either `n` or `nnn` are bound in more than one nested function. The compiler understands that, though.

anonymous function:
: A function that is not *bound* to a name.

applicative style:
: Applying a function to multiple lifted arguments using `map` and `apply` operators that make the code look parallel to a simple function application.

```
negate  $        1 + 2   --      -3
negate <$> Just (1 + 2)  -- Just -3
```

apply (a function):
: A function is applied to *arguments* to produce a value. This jargon is used where other language communities would speak of "calling a function" or "sending a message."

argument (to a function):
: An actual value given when a function is *applied*. Distinct from *parameter*, which is part of the function's definition.

associative (operator):

An operator is associative if, for any values `a`, `b`, and `c`, `(a op b) op c` is the same as `a op (b op c)`.

beta reduction:

You can simplify a function by replacing parameter names in its body with its arguments. So these two are the same:

```
(\n -> n + n) 5
5 + 5
```

bind (a name to a value):

Values (including function values) don't inherently have names. Instead, names refer to, or point to, values. A single value may have several names pointing to it. The process of associating a name with a value is called "binding."

bounded context:

"Bounded context" is a term from Domain-Driven Design, first described in Eric Evans's *Domain-Driven Design*. A bounded context is a collection of related software that shares a single model (which is compatible with, but not necessarily identical to, other models that deal with the same concepts). Bounded contexts translate data from the outside world into their model, and are reponsible for maintaining internal consistency.

canvas (SVG):

The place where SVG shapes are drawn. SVG canvases are embedded inside HTML.

characterization tests:

Tests used to make clearer, via examples, what a function or library does.

commutative:

A two-argument function is commutative if the arguments can be flipped without changing the result.

```
> (+) 1 2
3 : number
> (+) 2 1
3 : number
```

compose (functions):

When you compose two functions, the result of one is becomes the (sole) argument of the other. Alternately: you create a new function that behaves that way for its argument.

concrete type:

A type that contains no *type variables*. `String` is a concrete type, as is `(List (Maybe Int))`. `(List (Maybe a))` would not be.

constructor (data) :
Same as value constructor.

constructor (type):
A type constructor takes one type and produces another. For example, `Maybe` is a type constructor that can create the types `Maybe Int`, `Maybe String`, and so on.

constructor (value):
A function that creates a value "of" (matching) a particular type. `(Maybe a)` has two value constructors: `(Just a)` and `Nothing`.

constructor (smart):
A smart constructor is one that does some calculations on its arguments before calling a value constructor. They're often used to constrain arguments (for example, producing a `Nothing` if an `Int` argument isn't positive. Typically, only the smart constructors are exposed. Value constructors aren't, and so can only be used by the smart constructors, not client code.

continuation:
Any multi-step calculation within a function can be converted into one in which the result of the first step is passed to a single-argument function that represents the rest of the steps.

data constructor :
Same as value constructor.

Domain-specific language (DSL):
In the context of this book, a set of functions that can be composed to mimic the structure of a value to be created or processed. So the `Http` module contains a DSL for building HTTP structures: its functions are named after the tags they generate.

DSL: See *Domain-specific language*.

field (record):
A field is a name *bound* to a particular value in a record. In the following, `name` is a field of type `String`:

∼ { name = "Dawn" } – Elm syntax

easing function:
In an animation, the easing function takes a `Float` representing how much of the animation time has been spent, and produces a `Float` representing how much of the change has been completed. In an animation of a falling object, a quadratic easing function would make the object fall faster as it got closer to the ground.

eager evaluation:
In a language with eager evaluation, a function that returns a complex data structure calculates it fully when called, even if the caller doesn't use part of it. Contrast to lazy evaluation, in which only calculates parts when they're used.

effect (PureScript):
Data that identifies a non-pure function to call and what type of value it will return. Effects are synchronous.

eta conversion:
The following two expressions are equivalent and can be freely interchanged.

```
function
\arg -> function arg
```

This is convenient when `function` is a stage in a pipeline and you discover you want to the input value more than once. For example:

```
source |> \v -> (function v) + (function 2*v)
```

functor:
See TBD inside the book or outside it. Roughly, a type like `(List a)` or `(Maybe a)` that (1) can be asked for zero or more "wrapped" values, and (2) has a `map` or `fmap` function that follows specific "functor laws."

expose (a name):
Suppose we have two modules `Client` and `Source`. `Source` may choose to expose all, some, or none of its names. If a name is exposed, `Client` has access to the value it's *bound* to; otherwise not.

`Client` must *import* `Source` before it can use its names. By default, they'll be referred to as `Source.name`. If the client exposes one of the names, it can be referred to without the module name, just `name`.

higher-order function:
A function that takes another function as an argument, or one that produces a function as its return value.

hole (PureScript, Idris):
A `?hole` asks the compiler to tell you the type of any expression that can replace it. The compiler will also make some suggestions for an expression to use.

identity:
In a monoid, the identity is a value that can be appended to any value and produce the same value. The empty string is an identity for type `String`:

```
> "" ++ "foo"
"foo" : String
> "foo" ++ ""
"foo" : String
```

Also called “unit”.

immutability:
A value is immutable if it cannot be changed in a way that a program can observe. (The runtime can make changes “behind the scenes” so long as the program can’t detect it.) See the chapter on the topic.

import:
To make the names (and hence the values they’re *bound* to) in a `Server` module vailable to a `Client` module.

inferred type:
The type of a value that the compiler can compute from its definition. For example, the value of `List.map negate [1.0]` can be inferred to be `List Float` because `1.0` is a `Float`, `negate` produces a `Float` from a `Float`, and `List.map` produces a `List` of the type `negate` produces.

instance (of a type class):
A type that implements the functions and laws required of a type class.

label (record):
What Elm calls a record’s “field,” PureScript calls a “label.”

laws:
A set of equations describing a required relationship between functions. In a language with type classes, a type that defines law-abiding functions gets other correct functions automatically defined.

lazy evaluation:
In a language with lazy evaluation, a value is not calculated until some other calculation demands it. When you evaluate an expression in the repl (for example), the need to print the final value demands the values of subexpressions, which in turn demand the values of their subexpressions.

lens: A generic name for types that select a part from a whole. Lenses can be composed to form lenses that reach arbitrarily deeply. Lenses can be used both to retrieve parts and to change them.

lifting (a function):
Lifting converts a function in one “context,” “setting,” or “universe” into a corresponding

function within another. For example, `Maybe.map` can lift `uppercase` from a function that works on `String` to one that works on `Maybe String`.

macro:
A function describing a source-code to source-code transformation. The compiler applies macros to source it's compiling, then finishes compiling the final result.

module:
A named collection of *bindings* from names to values. A unit of encapsulation.

monoid:
If a type is a monoid, it has an associative function (`mappend`) to combine elements and produce a third of the same type. It also has an identity element (`mempty`) that can be appended to any value to produce the same value.

morphism:
A generalization of the idea of "function". A morphism takes a value of one type and produces a value of another type (or perhaps the same type).

mutability:
A mutable value can be changed ("mutated") by a program. The change can be observed via any of the names *bound* to it. See the chapter on the topic.

narrow (a type signature):
Provide, via a *type annotation*, a *type signature* more restrictive than the one the compiler *infers*. For example, the compiler might infer that a function *parameter* can be a `List anything`. By narrowing it to `List Int`, you instruct the compiler to reject other kinds of lists.

nominal typing:
Types defined only by their names and relationships to other type names. The structure of the data is irrelevant (unlike in structural typing).

opaque type:
A *sum type* whose *value constructors* are not *exposed*. Values of the type can only be created by calling other functions in the type's module. Often used to make it impossible for invalid values to be created.

open import:
An `import` statement that, by default, exposes all names in the imported module.

operator:
An operator is a function that can be used with infix notation, like `4 + 2`. Operators can be used as functions by surrounding them with parentheses: `(+) 4 2`. In some languages, a plain function can be used as an operator by surrounding it with backticks: `[1] ‘concat‘ [2]`.

parameter (of a function):
A name within a function definition. Values given during function *application* are called *arguments*.

partial application:
Giving fewer *arguments* than a function has *parameters*. Partial application produces a function that takes the remaining arguments and then, having all the arguments it needs, evaluates the body of the function.

partial function:
A function that doesn't handle some possible inputs. Calling it with one of those inputs has undefined results.

pattern matching:
Matching values with a "shape" against an expression of the same shape. Thus `(Maybe 5)` matches `(Maybe 5)` but not `Nothing`. The expression may contain variables. If the match succeeds, the variables are *bound* to the corresponding part of the value. Thus `(Maybe 5)` matches `(Maybe a)` and leaves `(a)` bound to `5`.

phantom type:
A *quantified type* at least one of whose *type variables* is used in none of the type's *value constructors*. Example:

```
type Tagged phantom value = Tagged value
```

See Generalizing tagging using phantom types for an extended example.

pipeline, **pipelining**, **processing pipeline**:
Computation by composing a number of single-argument functions and "flowing" values through the resulting pipeline. In Elm notation, this:

```
in |> f1 |> f2 |> f3
```

pipeline, **collection**:
Suppose you want to process every element of a collection like a list). For each element, there are several processing steps. When pipelining the collection, you apply the first processing step to all the elements, forming a new collection. Then you apply the next processing step to the elements of the new collection. Contrast to applying all the steps to the first element, then all to the second element, and so on.

product type:
A record is a product type. It's called that because the number of possible values is the product of the possible values of each of the *fields*.

property (`elm-style-animation`):
Either an SVG/HTML attribute or a CSS style.

property testing :

A property-based test uses a generator that creates a number of random values to submit to a test function. The creativity of the generator may (or may not) find test values that your own creativity didn't.

qualified name:

When referring to a value defined in another *module*, a qualified name includes both the module name (or an *alias*) and the name within the module.

```
... List.concat ...
```

quantified type:

A type that contains *type variables*.

```
type Maybe a = Just a | Nothing
```

Contrast to *concrete type*.

recursive function:

A function that can, in at least one of its possible evaluations, *apply* itself. It might be directly recursive, as when its own name appears within its definition, or indirectly, as when it applies a function that in turn applies it.

The smart-aleck glossary entry would look like this:

```
recursion:
  if you are tired of this definition then
    you understand recursion
  else
    see the entry for "recursion"
```

recursive type:

A type with at least one *value constructor* that refers to the type itself. Used to define the types of trees or other types of unbounded size.

```
type Node a = Construct a (Node a) | End
```

smart constructor:

A smart constructor is one that does some calculations on its arguments before calling a value constructor. They're often used to constrain arguments (for example, producing a `Nothing` if an `Int` argument isn't positive. Typically, only the smart constructors are exposed. Value constructors aren't, and so can only be used by the smart constructors, not client code.

structural equality:

Two values are equal if they are of the same type and all of their component parts are of the same type.

structural type (Elm):

The ability to describe a value with a subset of its fields. Consider the following:

```
{ record | name : String }
```

That requires a record with a `String` `name` field. It may also have other fields.

subscription (Elm):

A value that instructs the Elm runtime to convert a stream of zero or more external events into calls to the `update` function.

sum type:

A sum type describes a number of exclusive cases. It's called a "sum" type because the number of its possible values is the sum of the possible values for each of the cases.

tail recursion:

A recursive function such that at least path through it ends with a call to itself. A smart compiler can convert that path into a loop, which is considerably more space-efficient.

thunk:

A function with no arguments. In our languages, simulated with a single argument of type `Unit`.

total function:

A function that handles all possible inputs.

type alias (Elm):

A shorthand name that the compiler replaces with an actual type name.

```
type alias Percent = Tagged PercentTag Float
```

type class:

Suppose two types have some functions (and values) with the same names and type signatures. Suppose further that there's some set of laws that both types satisfy. They can then be declared to be of the same type class. That allows the compiler to pick the correct function based only on the shared name and the types of the arguments.

For example, the compiler uses `String.append` with `append "a" "b"` and `List.append` with `append [1, 2]`.

A type class can also contain functions defined in terms of the base function. Once a type defines the base functions, it gets the extra functions without nyone having to write any code.

type constructor:

A type constructor takes one type and produces another. For example, `Maybe` is a type constructor that can create the types `Maybe Int`, `Maybe String`, and so on.

tagged type:
: In this book, the use of sum types to add units to very generic types like `Int` or `Float` or `String`.

type annotation:
: A description of the type of a function or constant. It may be *narrower* than the type the compiler *infers*, but it may not allow values that Elm would not.

type signature:
: In this book, the type of a value, without caring whether it came from a *type annotation* or type *inference*.

type variable:
: A name in a type definition that stands in for a concrete type. Example:

```
type Maybe aTypeVariable = ...
```

unit (type):
: A type with a single value, also called unit. Both the type and the value are often written as empty parentheses: `()`.

unit (value):
: The identity element for an operator (that is, for a two-argument function).

value constructor:
: A function that creates a value "of" (matching) a particular type. `Maybe` has two value constructors: `Just` and `Nothing`. (As `Nothing` is a zero-argument function, it can be considered a constant.)

The singular truth about functions

I occasionally look at code and wonder what's really going on. For example, consider this from the discussion of `flip`:

```
> flip List.take [101, 202, 303] 2
[101,202] : List number
```

Because I think of `flip` as mainly a function that converts one function to another, I once found myself wondering "How does `flip` know to stop with `List.take`? Shouldn't the line have to look like the following?

```
> (flip List.take) [101, 202, 303] 2
[101,202] : List number
```

This appendix should help you answer that and similar questions.

If you find the answers fascinating and want to understand how far you can go with crazily simple functions, I highly recommend Raymond Smullyan's *To Mock a Mockingbird*. You will be amazed. Reginald Braithwaite has a nice free book adapting Smullyan's book to Ruby.

Single-argument functions

All functions, I claim, take a single argument. You might counter with the following function definition, which obviously takes two arguments:

```
sum x y = x + y
```

However, let's rewrite it using an anonymous function:

```
sum = \x y -> x + y
```

There are still two arguments. But there's a purely mechanical way to transform a two-argument function into two one-argument functions:

```
> sum = \x -> (\y -> x + y)
> sum 1 2
3 : number
```

We can step through why this function works by substituting actual arguments for `x` and `y`. Start with this:

```
> (\x -> (\y -> x + y)) 1 2
```

The outermost function is `(\x -> ...)`. It produces its value by substituting its argument (`1`) for `x`, producing this:

```
> (\y -> 1 + y) 2
```

So the result of applying the outer function is another function – which takes one argument. Since there's another argument available from the line, it gets substituted in:

```
> 1 + 2
3 : number
```

The fact that functions only take one argument is what makes partial application work. Consider `List.take`. Its real, behind-the-scenes definition is something like this:

```
take = \number -> (\list -> ...code that uses `number` and `list`...)
```

And so, a line like this:

```
> take2 = List.take 2
<function> : List a -> List a
```

... is really this:

```
> take2 = (\number -> (\list -> ...`number`...`list`...)) 2
```

The `2` is available to be substituted in, producing this:

```
> take2 = (\list -> ...2...`list`...)
```

Since there's nothing else to be substituted in, we're left with a function that takes a list.

Wait, why *doesn't* flip need to be parenthesized?

Given this:

```
> flip List.take [101, 202, 303] 2
```

… I wondered how the language “knew” to stop with the first argument. We’ve seen the answer now: not just `flip` but *every* function stops with the first argument. The line without parentheses is the same as the following fully-parenthesized form:

```
> ((flip List.take) [101, 202, 303]) 2
```

So when executed the line is, technically, creating a function from a function, using that second function to create a third function from a list, and using *that* to calculate a value from `2`. I write “technically” because a compiler is allowed to optimize code that doesn’t play partial application tricks.

How point-free style works

Let’s be more precise about how the following two different ways of defining `take2` mean the same thing:

```
take2 list = List.take 2 list
take2 = List.take 2
```

Just a few paragraphs ago, we saw that `List.take` has an behind-the-scenes definition of this form:

```
\number -> (\list -> ...code that uses `number` and `list`...)
```

… and we saw `List.take 2` simplifies, by substitution, to this:

```
(\list -> ...2...list...)
```

So the following two are equivalent:

```
take2 = List.take 2
take2 = (\list -> ...2...list...)
```

Now let’s look at the other definition:

```
take2 list = List.take 2 list
```

That’s the same as this:

```
take2 = \list -> (List.take 2) list
```

We can replace `List.take 2` with `(\list -> ...2...list...)`. The only snag is that we've two forms that use the variable `list`. I'll rename `List.take`'s version `list2`, giving us:

```
take2 = (\list -> (\list2 -> ...2...list2...)) list
```

We can substitute the *argument* `list` for the *parameter* `list`:

```
take2 = \list -> ...2...list...
```

And we've arrived at the same definition for `take2`, but from a starting point. QED, as they used to say in the old days.

For fun: William James’s story of the squirrel

From *Pragmatism*, 1905.

SOME YEARS AGO, being with a camping party in the mountains, I returned from a solitary ramble to find every one engaged in a ferocious metaphysical dispute. The corpus of the dispute was a squirrel – a live squirrel supposed to be clinging to one side of a tree-trunk; while over against the tree’s opposite side a human being was imagined to stand. This human witness tries to get sight of the squirrel by moving rapidly round the tree, but no matter how fast he goes, the squirrel moves as fast in the opposite direction, and always keeps the tree between himself and the man, so that never a glimpse of him is caught. The resultant metaphysical problem now is this: Does the man go round the squirrel or not? He goes round the tree, sure enough, and the squirrel is on the tree; but does he go round the squirrel? In the unlimited leisure of the wilderness, discussion had been worn threadbare. Every one had taken sides, and was obstinate; and the numbers on both sides were even. Each side, when I appeared therefore appealed to me to make it a majority. Mindful of the scholastic adage that whenever you meet a contradiction you must make a distinction, I immediately sought and found one, as follows: “Which party is right,” I said, “depends on what you practically mean by ‘going round’ the squirrel. If you mean passing from the north of him to the east, then to the south, then to the west, and then to the north of him again, obviously the man does go round him, for he occupies these successive positions. But if on the contrary you mean being first in front of him, then on the right of him, then behind him, then on his left, and finally in front again, it is quite as obvious that the man fails to go round him, for by the compensating movements the squirrel makes, he keeps his belly turned towards the man all the time, and his back turned away. Make the distinction, and there is no occasion for any farther dispute. You are both right and both wrong according as you conceive the verb ‘to go round’ in one practical fashion or the other.”

Although one or two of the hotter disputants called my speech a shuffling evasion, saying they wanted no quibbling or scholastic hair-splitting, but meant just plain honest English ‘round’, the majority seemed to think that the distinction had assuaged the dispute.